Design and development of a game framework

Bachelor Degree in Informatics Engineering
Bachelor thesis
June 29th, 2016

Author: Alessio Linares Bernardini
Director: Guillem Godoy
Specialization: Computer Science

Barcelona School of Informatics
Polytechnic University of Catalonia
# Contents

1 Listings

1 Preamble 5

2 Context 6

3 State of the art 7

4 Formulation of the problem 8

5 Objective 9

6 Scope 10

7 Anatomy of a game 11
   7.1 Actors and Scenes 12

8 Project implementation 13
   8.1 Evolution of the implementation 13
      8.1.1 Removal of Scene 13
      8.1.2 Vector2 and Vector3 13
      8.1.3 Draw using OpenGL 14
   8.2 Hummingbird 14
      8.2.1 Game 14
      8.2.2 Actor 16
      8.2.3 Behavior 18
      8.2.4 Transformation 19
      8.2.5 Time and Clock 19
      8.2.6 Logging 20
      8.2.7 Plugin 21
      8.2.8 Kinematic 22
      8.2.9 KinematicWorld 23
      8.2.10 Exceptions 23
      8.2.11 Vector2 23
## Contents

8.2.12 Vector3 ................................................................. 24
8.2.13 ResourceManager .................................................... 24
8.3 Multimedia OpenGL ................................................. 25
8.3.1 MultimediaOGL ..................................................... 25
8.3.2 Camera ................................................................. 27
8.3.3 Rectangle ............................................................... 28
8.3.4 Sprite ........................................................----------- 28
8.3.5 AnimatedSprite ...................................................... 30
8.3.6 InputHandler .......................................................... 31
8.3.7 ResourceManagers ................................................. 32

9 Game implementation example ....................................... 37
9.1 Player ................................................................... 37
9.2 Enemy ................................................................. 43
9.3 Bullet ................................................................. 49
9.4 Enemy Spawner ........................................................ 52
9.5 main.cpp ................................................................. 53
9.6 Resources .............................................................. 54
9.7 Math ................................................................. 56

10 Future work .............................................................. 57

11 Project planning ......................................................... 58
11.1 Methodology and rigor ............................................... 58
11.2 Description of tasks .................................................. 59
11.2.1 Project planning .................................................... 59
11.2.2 Game framework ................................................... 59
11.2.3 Multimedia OpenGL plug-in ..................................... 60
11.2.4 Final task ............................................................ 61
11.3 Time table ............................................................... 61
11.4 Gantt chart .............................................................. 62
11.5 Action plan ............................................................ 64
11.6 Budget estimation ..................................................... 64
11.6.1 Hardware resources ............................................... 64
11.6.2 Software resources ............................................... 65
11.6.3 Human resources ................................................ 65
11.6.4 Total budget ......................................................... 66
11.6.5 Budget control ..................................................... 66
11.7 Sustainability .......................................................... 66
11.7.1 Economical sustainability ........................................ 66
11.7.2 Social sustainability .............................................. 67
11.7.3 Environmental sustainability .................................... 67
11.7.4 Sustainability table ............................................... 68

12 Conclusions .............................................................. 69
Appendix 72
### Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Basic main loop</td>
<td>12</td>
</tr>
<tr>
<td>8.1 Creation and destruction of an Actor</td>
<td>16</td>
</tr>
<tr>
<td>8.2 Plugin usage example</td>
<td>16</td>
</tr>
<tr>
<td>8.3 Actor example</td>
<td>17</td>
</tr>
<tr>
<td>8.4 Behavior example</td>
<td>18</td>
</tr>
<tr>
<td>8.5 Transformation example</td>
<td>19</td>
</tr>
<tr>
<td>8.6 Clock and Time example</td>
<td>19</td>
</tr>
<tr>
<td>8.7 assert_msg() example</td>
<td>20</td>
</tr>
<tr>
<td>8.8 log() example</td>
<td>21</td>
</tr>
<tr>
<td>8.9 log_e() example</td>
<td>21</td>
</tr>
<tr>
<td>8.10 log_d() example</td>
<td>21</td>
</tr>
<tr>
<td>8.11 Plugin example</td>
<td>22</td>
</tr>
<tr>
<td>8.12 Kinematic example</td>
<td>22</td>
</tr>
<tr>
<td>8.13 KinematicWorld example</td>
<td>23</td>
</tr>
<tr>
<td>8.14 Vec2 example</td>
<td>24</td>
</tr>
<tr>
<td>8.15 Vec3 example</td>
<td>24</td>
</tr>
<tr>
<td>8.16 ResourceManager example</td>
<td>25</td>
</tr>
<tr>
<td>8.17 MultimediaOGL example</td>
<td>26</td>
</tr>
<tr>
<td>8.18 Camera default settings</td>
<td>27</td>
</tr>
<tr>
<td>8.19 Rectangle example</td>
<td>28</td>
</tr>
<tr>
<td>8.20 Sprite example</td>
<td>28</td>
</tr>
<tr>
<td>8.21 Astronaut walking animation</td>
<td>31</td>
</tr>
<tr>
<td>8.22 InputHandler example</td>
<td>31</td>
</tr>
<tr>
<td>8.23 TextureManager example</td>
<td>32</td>
</tr>
<tr>
<td>8.24 SpriteAnimationManager example</td>
<td>32</td>
</tr>
<tr>
<td>8.25 SoundManager example</td>
<td>33</td>
</tr>
<tr>
<td>8.26 MusicManager example</td>
<td>34</td>
</tr>
<tr>
<td>8.27 ShaderProgramManager example</td>
<td>34</td>
</tr>
</tbody>
</table>
1. Preamble

Through this document, the various functionalities of the resulting product, the Hummingbird framework, are explained with examples. For a complete description of each class and its methods a reference manual is included appended at the end of this document.

The source code for both, Hummingbird and Multimedia OpenGL (MOGL) Plugin, is open source under the MIT license and can be found at https://github.com/Galbar/hummingbird and https://github.com/Galbar/hummingbird-MOGL. For the exact version of the project described in this document refer to the TFG branch of the git repositories.
2. Context

My interest in game development started almost at the same time as my interest in programming.

First, I started experimenting with Game Maker\(^6\), where I was able to easily make games by basically drag-and-dropping icons that represented changes in the state of the game. After a lot of trial and error and messing with its scripting language too, I decided to try another tool: Unity Game Engine. Unity offered a more complex environment but it also allowed for more complex things. I started learning it by doing tutorials and eventually made a few mini-games.

At that point I wanted to implement games in a much lower level. I wanted to understand the insides of how a game works. I started programming games in C++ without using any game developing framework or engine (only using SFML\(^4\) for multimedia and rendering). The first time was very complicated, but the more games I developed, the better I saw a common pattern in their structure. I always had a main loop\(^8\) with a game state update and render; some kind of object pool; some kind of input handling; and some kind of rendering pipeline. On the other hand, I was almost never able to reuse them because I like trying new technologies, libraries or methods, and most of the time these common structures were entangled with calls to the specific libraries I was using at the moment.

Eventually, I told to myself, why not implement all these functionalities once, in a way that is reusable for my future projects? This is where this project comes into place, to implement a flexible and extensible game development framework.
3. State of the art

The functionalities of the game framework I want to develop for this project do not conform a full game engine, but they are just a part of what a game engine has to offer. For this reason, I considered convenient to study some commercial game engines. There exist generic game engines and game engines specialized in specific genres. I focused on the generic ones, such as Unity[10] and Unreal Engine [3], because one of my goals is to use it for different projects.

Both mentioned engines use componentization for giving their actors (game objects in Unity) behavior and properties [2][9]. Unity also implements a main loop with a fixed update[11], that is better for AI and physics simulations, as it provides stability and allows for simplicity [8]. On the other hand it is good to render as fast as you can and Unity has the rendering outside of the fixed update.

They also implement some kind of resource management that allows easy access to resources without having multiple copies of the same, which is a basic functionality for preventing filling the memory with unneeded data.

I also read the on-line book Game Programming Patterns[7], by Bob Nystrom, to get ideas and solutions to the problems I may face when implementing the framework.
4. Formulation of the problem

The problem I want to solve is having to implement the same infrastructure every time that I make a game. That infrastructure consists of having a main loop, a pool of game objects (each of them possibly having different behaviors) and handling their data.

This infrastructure, framework, must also be flexible enough to allow me to experiment with new technologies (rendering libraries, networking, physics, etc.) in the future without requiring to change its code, thus making the development of game easier.
5. Objective

The goal of this project is to design and implement a game development framework that solves the problem described in the previous section. The framework will have solved the problem when it presents the functionalities listed below.

- The framework implements some basic helper classes for game development, that is, provides a way to describe transformations of objects, and tools for measuring and storing intervals of time;
- The framework implements a main loop with fixed update;
- The framework handles the creation, destruction and storage of game actors. These actors will contain behaviors that will define its actions and properties;
- The framework offers a way of handling resources (loading, accessing, freeing);
- The framework will allow extending the main loop with a plug-in system. This feature makes possible adding a rendering pipeline, physics simulation, etc. without having to touch the framework code.

Finally, for demonstrating the usage of the plug-in system. This project will also implement a plug-in that wraps the SFML library. The SFML plug-in will implement:

- The resource management of textures, music, sounds, etc. using the structure offered by the framework;
- An input handling system;
- A 2D rendering pipeline using the SFML utilities.
6. Scope

This project is part of a bigger project that consists of developing a full game engine. Due to time constraints the project will be limited to just the core of the game engine. This core will consist of the framework that results of this project and that will be limited to the functionalities listed in the objectives above. For the sake of simplicity and avoiding possible complications, the framework spatial system will be limited to the 2 dimensional plane.

Moreover, the project does not consist of developing a game nor it is part of it. It’s sole goal is to develop the game framework.
7. Anatomy of a game

A video game is an electronic game that involves human interaction with a user interface to generate visual feedback on a video device such as a TV screen or computer monitor. — Wikipedia

The quote above is Wikipedia’s definition for video game. From it we can extract the various parts that form a video game:

• It’s a game;
• Involves human interaction (input);
• From the input generate some feedback;
• The feedback is visual (video).

In this document we won’t tackle the topic of what makes a game a game. Instead we’ll focus on the other three points that are specific to video games.

The first thing to clear up is that a video game, game from now on for simplicity, consists of an input-feedback loop. If it wasn’t a loop, the game would only last one frame.

We’ll call this input-feedback loop main loop or game loop and in its simplest form it looks like the following:
7.1. **Actors and Scenes**

```c
while (true) {
    processInput(); // Get the human interaction
    update(); // generate the feedback
    render(); // display the feedback
}
```

**Listing 7.1: Basic main loop.**

This is an extremely simplified version of a game loop. More complex versions may include sleep time between cycles, to lower the power consumption, or include a fixed cycle rate, which is useful for physics and AI simulation.

**Actors and Scenes**

As games grow more complex, proper structuration of data becomes more important, not only for efficiency but mainly for easing the development process. Generally, this structure is given by the division of the game in **scenes** and the representation of the game world through **actors** (or game objects).

Scenes are the natural division of a game. A scene can be a level, a menu screen, etc. Usually, they set up the configuration needed for that given part of the game, when they start, and, when done, they handle the clean up.

Actors, on the other hand, are the objects that live in the game world. This is a general concept including, not only the player, but also, for example, a rock blocking the player’s path. They, as their name suggests, *act* in the game’s world simulation.
8. Project implementation

The following sections will explain the evolution of the project and the available classes and their jobs in the final product. Refer to the reference manual appended at the end of the document for a full and detailed list of all the functionalities of each class.

Evolution of the implementation

Throughout the development of the project the design of the framework has suffered changes. A few of them are explained below.

Removal of Scene

The initial design of the framework included a class for scenes. After a few iterations the Scene class felt pointless, having the only functionality of creating the required actors at the start, keeping track of them and finally destroying them.

This functionality can be implemented, with better results, with a plugin that does exactly the same, and with the advantage that this plugin would be tailored to whatever needs the actual implementation of the game uses (p.e. scene file format).

And so, the Scene class was dropped.

Vector2 and Vector3

Initially there were not going to be any kind of vector class. The Transformation class would include all needed spatial information and that was it.

Once the implementation of MOGL started, the need for a way to represent spatial infor-
mation came into place. Mainly to normalize the representation of such information, at one point there were sf::Vector3 and glm::vec3 in the public API of the library.

The decision was made to implement Vector2 and Vector3 in an effort to unify the spatial information throughout all the Hummingbird framework and MOGL plugin.

**Draw using OpenGL**

The initial plan was to make use of SFML’s Drawable classes to render. At some point the initial desition was changed to render the geometry directly using OpenGL in order to make future implementations of Drawables for 3D geometry easier.

This change implied a complete refactor of the rendering pipeline, requiring the management of shaders and the addition of a camera.

**Hummingbird**

Hummingbird is the name of the base framework. In this section I’ll explain how to use the functionalities it features. The classes and methods that compose the framework are all contained inside the `hum (hummingbird)` namespace.

For more information on each of the classes and methods check the `hum namespace` section of the reference manual appended to this document.

**Game**

The Game class handles the game loop and the lifecycle of the game. To start running a Game we just to call `run()` and the Game instance will enter the following flow chart. The flow chart shows the lifecycle of a Game and the different steps that happen through it.
A Game instance can be queried for Time information at any step (when the Game is run-
8.2. Hummingbird

This information can be the deltaTime(), which is the Time that has passed between the previous update and the current update; the fixedUpdateTime() which is the Time that passes between fixedUpdate and fixedUpdate and is calculated from the fixed_tickrate constructor parameter; and the fixedUpdateLag(), which is the Time since the last fixedUpdate.

The running state of the Game can be controlled by using setRunning(). If set to false the game loop will exit and the Game will end. A Game instance should not be reused once the Game is done running, as the final state is not guaranteed in any way.

The Game class also owns the Actor object pool and therefore handles the creation and destruction of Actors. Example code:

```cpp
hum::Game game;
hum::Actor* new_actor = game.makeActor(); // creation of a new Actor;
// ...
game.destroy(new_actor); // mark the Actor to be destroyed.
```

Listing 8.1: Creation and destruction of an Actor

Actors are not destroyed right away, but just marked to be destroyed, and are finally destroyed after Plugin::postUpdate(). All Actors are destroyed automatically after Plugin::gameEnd().

A Game instance can also contain Plugins. Plugins can implement functionality for the Game such as a rendering pipeline, scene management, etc. They can be added and queried by typename using addPlugin() and getPlugin() template methods respectively (example below).

```cpp
class MyPlugin : public hum::Plugin {}

hum::Game game;
MyPlugin* mp = game.addPlugin<MyPlugin>();

// somewhere else in the code (p.e. inside a Behavior)
MyPlugin* mp = game().getPlugin<MyPlugin>();
```

Listing 8.2: Plugin usage example

A Plugin shouldn’t be added after calling run().

**Actor**

This class represents an object in the Game. You can create a new Actor by calling Game::makeActor(). This method will create a new Actor and return it. The Game owns the Actor and it controls its lifetime.

To destroy an Actor you must call Game::destroy(), not its destructor. The Actor then, will be marked to be destroyed and after the next update step it’ll be deleted. Just before being deleted, the Actor will call its Behaviors onDestroy() method.
An Actor, by default, doesn’t have any behaviour and you should not inherit from it. Instead Actors are composed by Behaviors. These Behaviors are the ones that must implement the behaviour of the Actor composed by them.

On the other hand, Actors do have a Transformation, accessible through transform() and a reference to its Game, accessible through game().

An Actor can be active or inactive. If a Actor is inactive it exists, and all its Behaviors also exist and have been instantiated; but it won’t be updated. The same applies to its Behaviors.

Usage example. The Actor will be destroyed after 10 fixed updates:

```cpp
// We define two Behaviors: A and B.

class B : public hum::Behavior
{
public:
    B(int x): value(x)
    {
        int value;
    }

class A : public hum::Behavior
{
public:
    A(int x): current(x)
    {
        void init() override
        {
            last = actor().getBehavior<B>().value;
        }

        void fixedUpdate() override
        {
            current++; 
            if (current > last)
            {
                actor().game().destroy(actor());
            }
            hum::log("Count: ", current);
        }

private:
    int last, current;
}

int main()
{
    hum::Game g;
    hum::Actor* a = g.makeActor();
    // here we add the A and B to the Actor.
    A* t = a->addBehavior<A>(1);
    a->addBehavior<B>(10);
```
Behavior

Class from which inherit to implement and give an Actor behavior.

A Behavior always lives inside an Actor. Said actor takes care of the lifecycle of the Behavior.

For creating a custom Behavior you may inherit from this class and override the methods you need to implement the wanted functionality.

Behaviors must also implement a static const char* behaviorName() method that, as the name hints, returns the class name. This is used for error reporting.

Usage example:

class PrintTransform : public hummingbird::Behavior
{
public:
    void init() override
    {
        hummingbird::log("Behavior initialized");
    }

    void fixedUpdate() override
    {
        hummingbird::log("Actor transformation: ", actor().transform());
    }

    void onDestroy() override
    {
        hummingbird::log("Actor destroyed");
    }

    static const char* behaviorName()
    {
        return "PrintTransform";
    }
};

Listing 8.4: Behavior example.
Transformation

hum::Transformation is a simple class that defines a spacial transformation of an object. That is it defines a 3D translation, rotation and scale, using hum::Vector3.

The hum::Transformation class has a small and simple interface, its position, rotation and scale members can be accessed directly (there are no accessors like setPosition(), getPosition()) and it contains no mathematical function other than the method transform(), which accumulates transformations.

Usage example:

```cpp
1  hum::Transformation t, t2;
2  t.position.x = 10.f;
3  t.rotation.z = 90.f;
4  t.scale.x = 0.5f;
5
6  t2.position.x = 5.f;
7  t2.rotation.z = -25.f;
8  t2.scale.x = 0.2f;
9
10 t = t.transform(t2);
11 hum::log(t); // hum::Transformation (position=hum::Vector3(15, 0, 0); rotation=hum::
12   Vector3(0, 0, 65); scale=hum::Vector3(0.1, 1, 1))
```

Listing 8.5: Transformation example

Time and Clock

The Time class represents a time interval. It has nanoseconds precision and can queried for its value in seconds, milliseconds, microseconds and nanoseconds.

The Clock class measures time intervals and stores them in an instance of Time. Clock too has nanoseconds precision.

Usage example:

```cpp
1  hum::Clock clk;
2  while (game_is_running)
3  |
4  hum::Time deltaTime = clk.reset();
5  ... // Game code
6 }
```

Listing 8.6: Clock and Time example
8.2. **Hummingbird**

**Logging**

The framework includes a set of methods to help the debug process by allowing to print messages depending on the environment (release or debug) and to various channels (standard or error).

**assert_msg()**

Check a condition and if it fails, print a given message and exit the program.

```cpp
template<typename ... TT>
inline void assert_msg(bool condition, const TT& ... tt)
```

The message can contain any type with the `operator overloaded or any of the following classes:

- `hum::Vector2`
- `hum::Vector3`
- `hum::Transformation`
- `hum::Time`
- `hum::Clock`

This method does nothing if NDEBUG is defined.

Usage example:

```cpp
hum::assert_msg(player_x > 64, "Player is outside of the map! x=", player_x);
```

Listing 8.7: assert_msg() example

**log()**

Print a message to the standard output.

```cpp
template<typename ... TT>
inline void log(const TT& ... tt)
```

`TT` can contain any type with the `operator overloaded or any of the following classes:

- `hum::Vector2`
8.2. Hummingbird

- `hum::Vector3`
- `hum::Transformation`
- `hum::Time`
- `hum::Clock`

Usage example:

```cpp
hum::log("Player position: ", actor().transform().position);
```

Listing 8.8: log() example

**log_e()**

Similar to log(), but the message is printed to the error output.

Usage example:

```cpp
hum::log_e("Player position: ", actor().transform().position);
```

Listing 8.9: log_e() example

**log_d()**

Similar to log() but, it does nothing if NDEBUG is defined.

Usage example:

```cpp
hum::log_d("Player position: ", actor().transform().position);
```

Listing 8.10: log_d() example

**Plugin**

Class from which inherit to implement and give a Plugin for the Game.

A Plugin always lives inside the Game. The Game takes care of the lifecycle of the Plugin.

For creating a custom Plugin you may inherit from this class and override the methods you need to implement the wanted functionality.

For more information on the lifecycle of a Game see the Game class description.

Usage example:
8.2. Hummingbird

```cpp
class DeltaTimePlugin : public hum::Plugin
{
    void gameStart() override {
        hum::log("Game just started");
    }

    void preUpdate() override {
        hum::log("delta time: ", game().deltaTime());
    }

    void preFixedUpdate() override {
        hum::log("fixed update lag: ", game().fixedUpdateLag());
    }

    void postFixedUpdate() override {
        hum::log("fixed update lag: ", game().fixedUpdateLag());
    }

    void postUpdate() override {
        hum::log("delta time: ", game().deltaTime());
    }

    void gameEnd() override {
        hum::log("Game just finished");
    }
}
```

Listing 8.11: Plugin example

### Kinematic

Add Kinematic properties to the Actor. (Requires KinematicWorld).

This Behavior allows to give a velocity and an acceleration to an Actor. This way the Actor’s Transformation will change automatically overtime following a kinematic movement.

Usage example:

```cpp
hum::Game game;
game.addPlugin<hum::KinematicWorld>();

hum::Actor* actor = game.makeActor();
hum::Kinematic* k = actor->addBehavior<hum::Kinematic>();
k->velocity().position.x = 5;
k->acceleration().rotation.z = 2;
```

Listing 8.12: Kinematic example
8.2. Hummingbird

**KinematicWorld**

Plugin that handles the transformation (movement, scale or rotation) of an Actor with a Kinematic behavior.

Usage example:

```cpp
hum::Game game;
game.addPlugin<hum::KinematicWorld>();
```

Listing 8.13: KinematicWorld example

**Exceptions**

**BehaviorNotFound**

Exception thrown when getting a Behavior from an Actor that does not contain it. (see Actor::getBehavior())

**PluginNotFound**

Exception thrown when getting a Plugin from a Game that does not contain it. (see Game::getPlugin())

**Vector2**

`hum::Vector2<T>` is a simple template class that defines a mathematical vector with two coordinates (x and y). It can be used to represent anything that has two dimensions: a size, a point, a velocity, etc.

The template parameter T is the type of the components of the vector. It can be any type that supports arithmetic operations (+, -, /, *) and comparisons (==, !=), for example int or float.

You generally don’t have to care about the templated form (`hum::Vector2<T>`), the most common specializations have special typedefs:

- `hum::Vector2<float>` is `hum::Vector2f`
- `hum::Vector2<int>` is `hum::Vector2i`
The hum::Vector2 class has a small and simple interface, its x and y members can be accessed directly (there are no accessors like setX(), getX()) and it contains no mathematical function like dot product, cross product, length, etc.

Usage example:
```cpp
hum::Vector2f v1(16.f, 24.f);
v1.x = 18.2f;
float y = v1.y;

hum::Vector2f v2 = v1 * 5.f;
hum::Vector2f v3;
v3 = v1 + v2;

bool different = (v2 != v3);
```

Listing 8.14: Vec2 example

### Vector3

hum::Vector3<T> is very similar to hum::Vector2, the only one difference is that it has three dimensions (x, y and z) instead of two. It works the same way Vector2 does and implements the same operations.

As for Vector2, you generally don’t have to care about the templated form (hum::Vector3<T>), the most common specializations have special typedefs:

- hum::Vector3<float> is hum::Vector3f
- hum::Vector3<int> is hum::Vector3i

Usage example:
```cpp
hum::Vector3f v1(16.5f, 24.f, 13.f);
v1.x = 18.2f;
float y = v1.y;

hum::Vector3f v2 = v1 * 5.f;
hum::Vector3f v3;
v3 = v1 + v2;

bool different = (v2 != v3);
```

Listing 8.15: Vec3 example

### ResourceManager

Class that implements the generic functionality of a resource manager.
This template class has three type parameters, two of which are optional. The first is the type of the data to manage. The second one is the type of the key to identify the managed data (std::string by default). The third is the data needed to load the resource (std::string by default).

Usage example (manager for sf::Texture):

```cpp
class TextureManager : public ResourceManager<sf::Texture>
{
protected:
  sf::Texture* loadResource(const std::string& name) override
  {
    sf::Texture* resource = new sf::Texture();
    if (!resource->loadFromFile(name))
      return nullptr;
    return resource;
  }

  // ...

textureManager tm;

  if (!tm.load("cat", "cat.jpg")) {
    hum::log_e("Error loading cat.jpg");
  }
  if (!tm.load("dog", "dog.jpg")) {
    hum::log_e("Error loading dog.jpg");
  }

  sf::Texture* cat = tm.get("cat"); // get the texture
  tm.free("cat"); // unload the cat texture manually

  // when destroyed the resource manager will free all loaded resources.
```

Multimedia OpenGL

MOGL (Multimedia OpenGL) is a library that extends the Hummingbird to implement graphic and multimedia features using SFML and OpenGL. All classes in the library use the mogl namespace.

MultimediaOGL

Plugin that handles all the rendering pipeline, input and resource management.
The MultimediaOGL plugin is the class that groups all the functionalities included in MOGL and makes them accessible in the hum::Game.

**Rendering**

The plugin handles the creation of the sf::Window and the OpenGL context. The window is made accessible through window().

The plugin also handles the camera, that is accessible by calling getCamera(). To set and get the clear color setClearColor() and getClearColor() can be called.

The rendering of all active and enabled Drawables happens at every postUpdate. This may seem anti-intuitive because the position of the Actors is only updated every fixedUpdate (we’d be drawing the same frame multiple times), but MultimediaOGL does something smart: for all Drawables whose Actors have a hum::Kinematic behavior, the position of the Drawable will be interpolated using the kinematic information and the fixedUpdateLag. Also, AnimatedSprites’s animation is also updated when drawn. This way we can have an as smooth as possible view of the game world and a fixedUpdate for the game logic.

Finally, a game space to draw space transformation can be set by using setDrawSpaceTransform() (for more details see the method’s details).

**Resource Management**

The MultimediaOGL plugin owns an instance of each of the resource managers included with MOGL. This way one can access any of them at any point inside the game.

Example:

```cpp
class Player : public hum::Behavior
{
private:
    hum::Kinematic* k;
    mogl::AnimatedSprite* spr;
    mogl::SoundId sound_id;
    mogl::MultimediaOGL* mogl;
public:
    void init() override
    {
        sound_id = 0;
        // Get a pointer to the MultimediaOGL instance in the Game
        mogl = actor().game().getPlugin<mogl::MultimediaOGL>();
        k = actor().getBehavior<hum::Kinematic>();
        spr = actor().getBehavior<mogl::AnimatedSprite>();
        spr->setSpriteAnimation(mogl->spriteAnimations().get("player_idle"));
    }
};
```
void fixedUpdate() override
{
    ...
    if (k->velocity().x != 0 || k->velocity().y != 0)
    {
        // Get the animation “player_walking” from the resource manager for sprite animations
        auto anim = mogl->spriteAnimations().get("player_walking");
        if (anim != spr->spriteAnimation())
            spr->setSpriteAnimation(anim);

        // Play a sound using the sound manager
        if (mogl->sounds().get(sound_id) == nullptr)
        {
            sound_id = mogl->sounds()->play("steps", 75, true);
        }
    }
    else
    {
        auto anim = mogl->spriteAnimations().get("player_idle");
        if (anim != spr->spriteAnimation())
            spr->setSpriteAnimation(anim);
        if (mogl->sounds().get(sound_id) != nullptr)
        {
            mogl->sounds()->get(sound_id)->stop();
        }
    }
}

Listing 8.17: MultimediaOGL example

Camera

The Camera is the device through which the player views the world.

By default the camera is set to be orthogonal. It is placed at the point (0, 0, -1) and looks towards (0, 0, 1) with a viewport of 100 by 100. The (0, 0) is located at the top left corner with the x-axis growing to the right and the y-axis growing downwards.

This default configuration is equivalent to the following code fragment:

mogl::Camera cam;
cam.setZNear(0.1f);
cam.setZFar(1000.0f);
cam.setOrthogonal(0, -100, 100, 0);
cam.setPosition(hum::Vector3f(0, 0, -1));
cam.setCenter(hum::Vector3f(0, 0, 1));
cam.setUp(hum::Vector3f(0, -1, 0));

Listing 8.18: Camera default settings
8.3. Multimedia OpenGL

Rectangle

A rectangle-shaped 1x1 Drawable.

For different sizes use the scale in either its hum::Actor’s hum::Transformation or the Drawable’s hum::Transformation.

The following program draws a 20 by 20 square centered in a 100 by 100 area (camera defaults) scaled to 600 by 600.

```cpp
#include "hummingbird/hum.hpp"
#include "MOGL/MOGL.hpp"

int main(void)
{
    hum::Game game;
    auto mogl = game.addPlugin<mogl::MultimediaOGL>(sf::VideoMode(600, 600), "Rectangle example");

    auto actor = game.makeActor();
    actor->addBehavior<mogl::Rectangle>(sf::Color::Red);
    actor->transform().position.x = 40;
    actor->transform().position.y = 40;
    actor->transform().scale.x = 20;
    actor->transform().scale.y = 20;

    game.run();
    return 0;
}
```

Listing 8.19: Rectangle example

The result is shown in figure 8.2

Sprite

A textured rectangle-shaped 1x1 Drawable

For different sizes use the scale in either its hum::Actor’s hum::Transformation or the Drawable’s hum::Transformation.

The following program draws a 20 by 20 cat image centered in a 100 by 100 area (camera defaults) scaled to 600 by 600.

```cpp
#include "hummingbird/hum.hpp"
#include "MOGL/MOGL.hpp"

int main(void)
{
    hum::Game game;
```
```cpp
auto mogl = game.addPlugin<mogl::MultimediaOGL>(sf::VideoMode(600, 600), "Rectangle example");

// load the texture and let the manager handle its lifetime
mogl->textures().load("cat", "cat.jpg");

auto actor = game.makeActor();
actor->addBehavior<mogl::Sprite>(mogl->textures().get("cat");
actor->transform().position.x = 40;
actor->transform().position.y = 40;
actor->transform().scale.x = 20;
actor->transform().scale.y = 20;

return 0;
```

Listing 8.20: Sprite example

The result is shown in figure 8.3.

Figure 8.2: Rectangle example
8.3. Multimedia OpenGL

AnimatedSprite

A textured rectangle-shaped 1x1 Drawable that changes over time.

For different sizes use the scale in either its hum::Actor’s hum::Transformation or the Drawable’s hum::Transformation.

The AnimatedSprite plays the animation defined in the SpriteAnimation assigned to it.

For an example of usage of both AnimatedSprite and SpriteAnimation see the section of the SpriteAnimationManager8.24

SpriteAnimation

A SpriteAnimation models an animation built from a sequence of frames stored in a tilesheet.
Given the following tilesheet in figure [8.4] the SpriteAnimation that stores the walking animation of the astronaut is defined in the next code fragment.

```cpp
mogl::SpriteAnimation walk{
    // texture containing the tilesheet
game().getPlugin<mogl::MultimediaOGL>()->textures().get("sprites"),
    0, // Horizontal offset of the tilesheet
    0, // Vertical offset of the tilesheet
    0, // Horizontal margin between tiles in the tilesheet
    0, // Vertical margin between tiles in the tilesheet
    48, // Width of a tile in the tilesheet
    48, // Height of a tile in the tilesheet
    // Sequence of ids of the tiles in the tilesheet to play.
    {0, 1, 2, 3, 5, 6, 7, 8},
    // Sequence of hum::Times for each frame in the animation.
    std::vector(8, hum::Time::milliseconds(75))
};
```

Listing 8.21: Astronaut walking animation

**InputHandler**

Class that handles SFML input events and allows for easy querying of related data.

Usage example:

```cpp
mogl::InputHandler& input = game.getPlugin<mogl::MultimediaOGL>()->input();

// querying the keyboard
if (input.isKeyPressed(sf::Keyboard::Space))
    hum::log("Space has been pressed");

// querying the mouse
hum::Vector2i mouseDelta(0);
if (input.mouseMoved())
    mouseDelta = input.getMouseCurrentPosition() - input.getMousePreviousPosition();
```

Listing 8.22: InputHandler example
8.3. Multimedia OpenGL

Resource Managers

TextureManager

A hum::ResourceManager for sf::Texture.

Example:

```cpp
mogl::TextureManager tm;
tm.load("cat", "path/to/cat.jpg");
sf::Texture* cat = tm.get("cat");
//...
actor.addBehavior<mogl::Sprite>(cat);
//...
tm.free("cat");
```

Listing 8.23: TextureManager example

SpriteAnimationManager

A hum::ResourceManager for mogl::SpriteAnimations.

This Resource manager is different because it uses SpriteAnimations as the data to "load" and stores a copy of the given SpriteAnimation.

Example:

```cpp
mogl::SpriteAnimation jump {
    // texture containing the tile sheet
game().getPlugin<mogl::MultimediaOGL>().get("player"),
    0, // Horizontal offset of the tile sheet
    0, // Vertical offset of the tile sheet
    0, // Horizontal margin between tiles in the tile sheet
    0, // Vertical margin between tiles in the tile sheet
    48, // Width of a tile in the tile sheet
    48, // Height of a tile in the tile sheet
    {5, 6, 7, 8}, // Sequence of ids of the tiles in the tile sheet to play.
    std::vector(4, hum::Time::seconds(0.3f)) // Sequence of hum::Times for each frame in the animation.
};
mogl::SpriteAnimationManager sam;
sam.load("player_jump", jump);
//...
```
8.3. Multimedia OpenGL

actor.addBehavior(mogl::AnimatedSprite>(sam.get("player_jump"));
// ...
sam.free("player_jump");

Listing 8.24: SpriteAnimationManager example

SoundManager

A hum::ResourceManager for sf::SoundBuffers.

This Resource manager is different because it overwrites the get() method and has other extra methods.

The SoundManager not just loads sf::SoundBuffers but also allows to play them through the method play(). This method returns a std::pair containing the id of the sound resource used (the one that is playing the required sf::SoundBuffer) and a pointer to the sf::Sound managing the playback of the sf::SoundBuffer. Note that a sound resource is not the same as a sf::SoundBuffer.

The get() methods are also different. In this manager they are accessed by sound ids (which are returned by play) and they return a pointer of the given sf::Sound, or nullptr if the sound has been freed.

Example:

mogl::SoundManager sm;
sm.load("roar", "path/to/roar.ogg");
// ...
// fixedUpdate()
if (event)
{
    roar_id = sm.play("roar", 50).first; // start playing the "roar" sound
}
// onDestroy()
if (sm.get(roar_id) != nullptr) // if the sound is still playing

sm.free("roar");

Listing 8.25: SoundManager example
8.3. Multimedia OpenGL

As shown in the example, when a sf::SoundBuffer playback is done (Stopped), then the sound resource is cleared and therefore the sound id invalidated (getting its related sound returns nullptr).

Sound ids start at 1 and always grow, that means that a sound id of 0 will always return nullptr.

MusicManager

A hum::ResourceManager for sf::Music.

Example:

```cpp
mogl::MusicManager mm;
mm.load("music1", "path/to/music1.ogg");
sf::Music* music = mm.get("music1");
music->play();
//...
music->stop();
mm.free("music1");
```

Listing 8.26: MusicManager example

ShaderProgramManager

A hum::ResourceManager for mogl::ShaderProgram.

This Resource manager is different because it uses ShaderProgramDefs to load the resource (ShaderProgram) instead of the usual 'std::string'.

Example:

```cpp
// Create shaders
mogl::Shader vs, fs;

// Load vertex shader from file
vs.loadFromFile(Shader::Type::VERTEX_SHADER, "shader.vert");
if (!vs.isCompiled())
{
    hum::log_e("Vertex shader failed to compile: ", vs.log());
    return 1;
}

// Load fragment shader from file
fs.loadFromFile(Shader::Type::FRAGMENT_SHADER, "shader.frag");
if (!fs.isCompiled())
{
    hum::log_e("Fragment shader failed to compile: ", fs.log());
    return 1;
}
```
8.3. Multimedia OpenGL

Listing 8.27: ShaderProgramManager example

```cpp
mogl::ShaderProgramDef def { vs, fs, "out_color"};
mogl::ShaderProgramManager spm;
spm.load("plain_shader", def);

mogl::ShaderProgram* sp = spm.get("plain_shader");
sp->use()->setUniform4f("color", 0, 1, 1, 1);

// ...

spm.free("plain_shader");
```
Figure 8.4: sprites.png
This section explains the implementation of a simple game using Hummingbird. The code and resources for the game exposed can be found at [https://github.com/Galbar/space-shooter](https://github.com/Galbar/space-shooter).

The game, Space Shooter, consists of an astronaut that has to shoot alien enemies that spawn and survive.

### Player

This class implements the Behavior for the Player (the astronaut). It implements the movement, using the keyboard, and displays it on screen. The player has a health bar on top of it. The main problem is that we’ll be rotating the player actor (we want the health bar to remain unrotated) and thus, any Drawable attached to it will also rotate, therefore what we’ll do to display the health bar is to have another actor that mimics all movement of the player, except for the rotation. We’ll call this actor the helper.

Below the class code is exposed and explained.

```cpp
#ifndef PLAYER_HPP
#define PLAYER_HPP

#include "hummingbird/hum.hpp"
#include "MOGL/MOGL.hpp"
#include "Bullet.hpp"
#include "Resources.hpp"
#include "math.hpp"

class Player : public hum::Behavior
{
private:
    static int s_lives;
    static float s_vel;
    static unsigned int s_ms_shoot;
    int p_live;
    mogl::MultimediaOGL* p_mogl;
};
#endif
```
We start by including the required files. First, we include Hummingbird and MOGL. Then, the various files for the game that are used by the player: Bullet.hpp (the projectile shot by the player), Resources.hpp (includes some functions for loading configuration files) and math.hpp (includes cmath and implements the method square(float x), useful for calculating distances). We’ll see those files in more detail later.

Next, we define the Player class, which is a Behavior, and its private methods. The static variables are used for storing class wise configuration values, read from a configuration file
on the first instantiation of the class. Then we store useful information, such as the number of lives that the player has left and pointers to useful components such as the kinematic behavior of the player actor and the helper actor; MOGL, for accessing input information; the sprite of the player and the rectangle of the health bar; and a clock for measuring time, such as the time between shots.

```cpp
public:
    Player()
    {
        if (s_vel == -1)
        {
            std::stringstream ss;
            readFileContents("res/config/Player.cfg", ss);
            ss >> s_vel
               >> s_ms_shoot
               >> s_lives;
        }
    }
```

As commented before, the first time a Player is instantiated the configuration is read from a file. This avoids having to recompile the project for adjusting gameplay values.

Afterwards, the initialization function of the Behavior is defined. We use it to give the player actor all behaviors and configuration needed, such as adding the AnimatedSprite to it and creating the helper actor for the health bar.

```cpp
void init() override
{
    // get the MOGL plugin instance and store it
    p_mogl = actor().game().getPlugin<mogl::MultimediaOGL>();

    // get the SpriteAnimation, that has already been loaded before
    // game start, and create the AnimatedSprite with it.
    mogl::SpriteAnimation* player_animation =
        p_mogl->spriteAnimations().get("player");
    p_spike =
        actor().addBehavior<mogl::AnimatedSprite>(player_animation);
    // Set the animation to pause as the player is not moving.
    p_spike->pause();
    // fix the rotation of the sprite.
    p_spike->transform().rotation.z = -90;

    // Set the center for transformations of the sprite to the center
    // of the astronaut tile (a bit displaced from the actual center
    // of the 48 by 48 tile)
    p_spike->setOrigin(hum::Vector3f(24./48., 18./48., 0));

    // Add kinematic behavior to the player actor and store a pointer
    // to it
    p_kinematic = actor().addBehavior<hum::Kinematic>();

    // set other useful information
    p_prev_rotation = 0;
```
9.1. Player

```cpp
p_live = s_lives;

// Create the helper actor we don’t need to save a pointer to it, as we will mainly work with its Kinematic Behavior and said behavior has a reference to its actor.
hum::Actor* helper = actor().game().makeActor();
// Sync the transformations
helper->transform() = actor().transform(); // Add Kinematic, save it and sync it with the player’s one.
p_helper_kinematic = helper->addBehavior<hum::Kinematic>();
helper->transform().position.z = -0.5;
helper->transform().position.y -= 0.8;
helper->transform().position.x -= 0.4;
// Add the background of the health bar
auto rect =
    helper->addBehavior<mogl::Rectangle>(sf::Color::White);
rect->transform().scale = hum::Vector3f(0.8, 0.1, 0.02);
// Add the foreground color of the health bar (the actual indicator)
p_live_rectangle =
    helper->addBehavior<mogl::Rectangle>(sf::Color::Green);
p_live_rectangle->transform().scale =
    hum::Vector3f(0.8, 0.1, 0.02);
p_live_rectangle->transform().position.z = -0.1;
```

Now we need to implement the update method of the player. We will read the input using the InputHandler instance inside the MOGL plugin and update the player’s speed depending on it. We’ll also create Bullets if the user presses the left mouse button.

```cpp
void fixedUpdate() override
{
    if (isDead())
    {
        p_sprite->stop();
        p_kinematic->velocity().position.x = 0;
        p_kinematic->velocity().position.y = 0;
        p_kinematic->velocity().rotation.z = 0;
        p_helper_kinematic->velocity().position.x =
            p_kinematic->velocity().position.x;
        p_helper_kinematic->velocity().position.y =
            p_kinematic->velocity().position.y;
        return;
    }

    // Look at the mouse
    hum::Vector2f mouse = p_mogl->input().getMouseCurrentPosition();
    mouse /= 48.f;
```
9.1. Player

```c++
float x = mouse.x - actor().transform().position.x;
float y = mouse.y - actor().transform().position.y;
float angleInRadians = std::atan2(y, x);
float angleInDegrees = (angleInRadians / M_PI) * 180.0;
float delta = angleInDegrees - p_prev_rotation;
if (delta > 180)
    delta -= 360;
else if (delta < -180)
    delta += 360;
p_kinematic->velocity().rotation.z =
    delta / actor().game().fixedUpdateTime().asSeconds();
p_prev_rotation = angleInDegrees;
```

In the fragment above, we first get the current position of the mouse and we transform it to game world coordinates by dividing it by 48. This is because in the main function we set the window to be 1000 by 1000 pixels, but the viewport to display 1 by 1 squares as 48 by 48 pixels on screen. We do this because, by default, all Drawables are 1 by 1 of size in the game world. To avoid having to set scales to all of them, we scale the view.

Then, we calculate the angle from the player's position to the mouse position and, from it, the speed at which the player is rotating. This is where p_prev_rotation becomes useful.

```c++
// Movement
if (p_mogl->input().isKeyDown(sf::Keyboard::A))
{
    p_kinematic->velocity().position.x =
    -s_vel * (not p_mogl->input().isKeyDown(sf::Keyboard::D));
}
else if (p_mogl->input().isKeyDown(sf::Keyboard::D))
{
    p_kinematic->velocity().position.x = s_vel;
}
else
{
    p_kinematic->velocity().position.x = 0;
}
if (p_mogl->input().isKeyDown(sf::Keyboard::W))
{
    p_kinematic->velocity().position.y =
    -s_vel * (not p_mogl->input().isKeyDown(sf::Keyboard::S));
}
else if (p_mogl->input().isKeyDown(sf::Keyboard::S))
{
    p_kinematic->velocity().position.y = s_vel;
}
else
{
    p_kinematic->velocity().position.y = 0;
}
```
For the movement, we just query the keyboard for the status of the direction keys: A, S, D, W. And depending on the input, set the speed of the actor. Finally, if the player is moving, play the animation; pause it otherwise.

    For shooting we check if the mouse left button is pressed and if the minimum time between shots has passed. The time is checked by using the clock.

        // Shooting
        if (p_mogl->input().isMouseButtonDown(sf::Mouse::Left)
            and p_clock.getTime().asMilliseconds() >= s_ms_shoot)
        {
            auto bullet = actor().game().makeActor();
            float mod = sqrt(square(x) + square(y));
            x /= mod;
            y /= mod;
            bullet->addBehavior<Bullet>(x, y);
            bullet->transform().position.x =
                actor().transform().position.x + 0.6 * x - 0.14 * y;
            bullet->transform().position.y =
                actor().transform().position.y + 0.6 * y + 0.14 * x;
            p_clock.reset();
        }

    If the conditions for shooting are met, we create a new actor, add to it a Bullet Behavior, give a direction to it and place it in front of the player. Lastly, we reset the clock.

        p_helper_kinematic->velocity().position.x =
            p_kinematic->velocity().position.x;
        p_helper_kinematic->velocity().position.y =
            p_kinematic->velocity().position.y;
        sf::Listener::setPosition(
            actor().transform().position.x,
            actor().transform().position.y, 10);
    }

    At the end of the update we sync the helper kinematic and set the position of the listener (sounds).

    void hit()
Finally, we implement some functions allowing other classes to interact with the player and we initialize the static members.

### Enemy

This class implements the Behavior for the Enemy of the game. Its AI is defined as state machine, represented in figure 9.2 and thus the update method is divided in the various states it can be.

```cpp
#ifndef ENEMY_HPP
#define ENEMY_HPP
#include "hummingbird/hum.hpp"
#include "MOGL/MOGL.hpp"
#include "Resources.hpp"
#include "Player.hpp"
#include "math.hpp"

class Enemy : public hum::Behavior
{
  private:
    enum State { WALKING, HITTING, WAITING, WIN, DONE };
    static float s_vel;
    static float s_min_player_dist;
    hum::Clock p_clock;
    State p_status;
    Player* p_player;
    hum::Kinematic* p_kinematic;
    float p_prev_rotation;
    mogl::AnimatedSprite* p_sprite;
    mogl::AnimatedSprite* p_blood;
    mogl::MultimediaOGL* p_mogl;
#ifdef ENEMY_HPP
#endif
```
As in the Player class, we include the required files, define private variables to store useful
9.2. Enemy

information and use the constructor for reading the class configuration. Some of the private information is similar to the player’s one.

Enemy receives a pointer to the player in its constructor, we store it. We’ll use it to check if the player is in hitting range and if so hit it.

Unlike Player, Enemy has two AnimatedSprites. The first is the one that plays the various enemy animations: walking, hitting, winning; the second plays the blood animation, only show when the player is hit. Finally, the we store a SoundId to handle the playback of the roar sound that the Enemy can produce randomly when alive.

```cpp
void init() override
{
    p_kinematic = actor().addBehavior<hum::Kinematic>();
    p_prev_rotation = 0;

    p_mogl = actor().game().getPlugin<mogl::MultimediaOGL>();
    p_sprite = actor().addBehavior<mogl::AnimatedSprite>(p_mogl->spriteAnimations().get("enemy_walking"));
    p_sprite->pause();
    p_sprite->setOrigin(hum::Vector3f(24. / 48., 18. / 48., 0));
    actor().transform().rotation.z = -90;

    p_blood = actor().addBehavior<mogl::AnimatedSprite>(p_mogl->spriteAnimations().get("enemy_attack1_blood"));
    p_blood->setLooping(false);
    p_blood->stop();
    p_blood->disable();
    p_blood->setOrigin(hum::Vector3f(24. / 48., -10. / 48., 0));

    p_status = WAITING;
}
```

Like in Player, we use the initialization method for configuring the actor. We add the Kinematic Behavior and the AnimatedSprites, setting each of them accordingly. Finally, we set the initial AI state.

When the Enemy is destroyed, we check if it is playing the roar sound and, if so, we stop it.

```cpp
void onDestroy() override
{
    if (p_mogl->sounds().get(p_sound) != nullptr)
    {
        p_mogl->sounds().get(p_sound)->stop();
    }
}
```

Now we deal with the most interesting part: the update. We start by defining how the base state changes: if the Enemy is done, do nothing; and if the player dies, change to win state.

In the win state, the Enemy does nothing, just waits one second still. Therefore, we stop all animations and reset the clock to measure the second.
9.2. Enemy

```cpp
void fixedUpdate() override
{
    if (p_status == DONE)
    {
        return;
    }

    if (p_player->isDead() and p_status != DONE and p_status != WIN)
    {
        p_status = WIN;
        p_sprite->stop();
        p_blood->disable();
        p_clock.reset();
    }
}
```

Next, we calculate the rotation speed of the Enemy. This step is the same as the player, with the difference that the position to look at is not the one of the mouse but one of the player. We also calculate the distance between the enemy and the player and set the speed of the enemy to 0.

```cpp
// Look at the player
float x = p_player->actor().transform().position.x - actor().transform().position.x;
float y = p_player->actor().transform().position.y - actor().transform().position.y;
float angleInRadians = std::atan2(y, x);
float angleInDegrees = (angleInRadians / M_PI) * 180.0;
float delta = angleInDegrees - p_prev_rotation;
if (delta > 180)
{
    delta -= 360;
}
else if (delta < -180)
{
    delta += 360;
}
p_kinematic->velocity().rotation.z = delta / actor().game().fixedUpdateTime().asSeconds();
p_prev_rotation = angleInDegrees;
p_kinematic->velocity().position.x = 0;
p_kinematic->velocity().position.y = 0;

// distance between the enemy and the player
float mod = sqrt(square(x) + square(y));
```

If the enemy is in win state and the one second waiting time has passed, the enemy changes to done state and starts the win animation. We set the animation not to loop so that it stops when done.

```cpp
if (p_status == WIN)
{
    if (p_clock.getTime().asSeconds() > 1)
```
On the other hand, if the enemy is walking and the player is not in hitting range the enemy moves towards the player. We set the speed, play the walking animation if not playing already and, with a probability of 0.01, play the roar sound if the enemy is not already roaring. We store the SoundId of the roar sound so that we can check in future updates if the enemy is still roaring or not.

```cpp
else if (p_status == WALKING and s_min_player_dist < mod)
{
    x /= mod;
y /= mod;
p_kinematic->velocity().position.x = x * s_vel;
p_kinematic->velocity().position.y = y * s_vel;
    if (p_sprite->status() != mogl::AnimatedSprite::Status::PLAYING)
    {
        p_sprite->play();
    }
    if (p_mogl->sounds().get(p_sound) == nullptr and (rand() %100) > 99)
    {
        auto info = p_mogl->sounds().play("roar", 1000);
        info.second->setAttenuation(0.1);
        p_sound = info.first;
    }
    else
    {
        if (p_status != WAITING)
        {
            if (p_status != HITTING) // state == WALKING and player is in hitting distance
            {
                p_sprite->setSpriteAnimation(p_mogl->spriteAnimations().get("enemy_attack1");
                p_sprite->play();
                p_sprite->setLooping(false);
                p_status = HITTING;
            }
        }
    }
}
```

If the enemy is walking and the player is in hitting distance, the enemy changes to hitting state. Changes the animation of the sprite and sets it not to loop, this way we can check when the animation is done and change to the next state. Finally, we play the sound of the enemy’s attack and set its position and attenuation. In this case, we don’t need to store the SoundId because we know that the hit sound will finish before the enemy is able to hit again.
Enemy

sf::Sound* sound = p_mogl->sounds().play("enemy_attack", 50).second;

sound->setPosition(actor().transform().position.x, actor().
transform().position.y, 0);

| sound->setAttenuation(0.01);
| else
|}

If the state is hitting and the hitting animation has finished, the status changes to waiting. We set the animation back to walking and disable the blood AnimatedSprite, in case it was displayed. The clock is also resetted so that we can measure the waiting time.

if (p_sprite->status() == mogl::AnimatedSprite::Status::STOPPED)
{
  p_sprite->setSpriteAnimation(p_mogl->spriteAnimations().get("enemy_walking"));
  p_sprite->setLooping(true);
  p_status = WAITING;
  p_blood->stop();
  p_blood->disable();
  p_clock.reset();
}

Else, if the hitting animation hasn't finished but it just passed the frame where the hit happens, check if the player is still in range and if so, hit it and display blood.

else if (p_sprite->frameIndex() > 1 and not p_blood->isEnabled())
{
  if (mod < s_min_player_dist)
  {
    p_blood->enable();
    p_blood->play();
    p_player->hit();
  }
  }
}
else
{

Finally, if the enemy is in waiting state, check if the waiting time has passed and if so, set the state to walking.

hum::Time t = p_clock.getTime();

if (t.asSeconds() > 1)
{
  p_status = WALKING;
}
}
As a last detail, if the roar sound is being played, update its position.

```cpp
sf::Sound* sound = p_mogl->sounds().get(p_sound);
if (sound != nullptr)
    {
        sound->setPosition(actor().transform().position.x, actor().transform().position.y, 0);
    }

static const char* behaviorName()
    {
        return "Enemy";
    }
```

**Bullet**

This class implements the behavior of a bullet shot by the player. The bullet has a lifespan of half a second and if while alive it hits an Enemy, then it kills it and destroys itself.

```cpp
#define BULLET_HPP

class Enemy;
class Bullet : public hum::Behavior
{
private:
    static float s_vel;
    float p_comp_x, p_comp_y;
    hum::Clock clk;
    hum::Kinematic* p_kinematic;
    mogl::Sprite* p_bullet;
    mogl::AnimatedSprite* p_explode;

public:
    Bullet(float comp_x, float comp_y):
        p_comp_x(comp_x),
        p_comp_y(comp_y)
    {
        if (s_vel == -1)
            {
                std::stringstream ss;
                readFileContents("res/config/Bullet.cfg", ss);
                ss >> s_vel;
        
```
The Bullet stores its direction, Kinematic, the Sprite for displaying the actual bullet and an AnimatedSprite for playing the explosion when being destroyed.

As in previous Behaviors, we use the constructor to read the configuration only once and we also store the direction in which the bullet will move.

Then, on initialization, we again add the required Behaviors to the actor and set their values to properly display and move the bullet. First, we add the animation for the explosion of the bullet, we disable it so that it doesn’t display anything and we center it to the actor’s position. Next, we add the actual bullet sprite and, again, we center it and rotate it towards the direction of the movement. Following that, we add the Kinematic and set the speed and, finally, we play the gun_shot sound and reset the clock to measure the bullet’s lifetime.
The Bullet’s fixed update is quite simple. First, it checks if the bullet has lived over its lifespan and, if so, starts the explosion animation. Next, it checks if the explosion animation is over and destroys the bullet if so. Finally, it iterates through all the Actors in the game; checks if they are in hit range and if they are of type Enemy; and if they are enemies it kills the enemy and plays the explosion animation.
**Enemy Spawner**

EnemySpawner, as its name suggests spawns enemies. It spawns a new one every half second and positions it randomly inside the viewport.

We will implement the EnemySpawner using a Plugin. This way we don’t need to have an extra actor in the game (good for Bullet’s checks).

The implementation of the spawner is quite simple. In the constructor we store a pointer to the player, so that we can pass it to the Enemies it creates. Then in the `preUpdate`, if the player is dead does nothing, otherwise it creates a new Enemy every half second.

```cpp
#ifndef ENEMY_SPAWNER_HPP
#define ENEMY_SPAWNER_HPP
#include <cstdlib>
#include "hummingbird/hum.hpp"
#include "Enemy.hpp"

class Player;
class EnemySpawner : public hum::Plugin
{
public:
    EnemySpawner(Player* player)
    {}

    void preUpdate() override
    {
        if (p_player->isDead())
        {
            return;
        }
        if (clk.getTime().asSeconds() > 0.5)
        {
            auto enemy = game().makeActor();
            enemy->addBehavior<Enemy>(p_player);
            enemy->transform().position.x = rand() % (1000/48);
            enemy->transform().position.y = rand() % (1000/48);
            clk.reset();
        }
    }

private:
    Player* p_player;
    hum::Clock clk;
};
#endif
```
main.cpp

In the main function we create the Game and add the KinematicWorld and MultimediaOGL plugins to it, in this last one we set the size of the window and its title. Then, we load the resources from the resources configuration file.

Next, we set the camera. By default, Rectangle, Sprite and AnimatedSprite have a size of 1 by 1 space units. As the tiles we want to draw are 48 by 48 pixels, we set the camera viewport to show 21 by 21 space units in a 1000 by 1000 window, making 1 space unit to occupy approximately 48 pixels on-screen.

Then, we create an Actor, add it a Player Behavior and set its initial position.

Finally, we add the EnemySpawner to the game, start the background music and run the game.

```cpp
#include "hummingbird/hum.hpp"
#include "MOGL/MOGL.hpp"
#include "Resources.hpp"
#include "Player.hpp"
#include "EnemySpawner.hpp"

int main()
{
    hum::Game g(60);
g.addPlugin<hum::KinematicWorld>();
mogl::MultimediaOGL* mogl = g.addPlugin<mogl::MultimediaOGL>(sf::VideoMode(1000, 1000), "Space Shooter");
loadResources(*mogl, "res/resources.def");

    // 21 ~= 1000/48
    mogl->getCamera().setOrthogonal(0, -21, 21, 0);

    auto a = g.makeActor();
    auto player = a->addBehavior<Player>();
a->transform().position.x = 10;
a->transform().position.y = 10;

    g.addPlugin<EnemySpawner>(player);
sf::Music* music = mogl->music().get("ambiance");
music->setLoop(true);
music->play();
g.run();
return 0;
}```
Resources

This file contains a set of fancy methods so that I can define the game configuration outside of the code, so that recompilation of the game isn’t needed when changing a setting.

```cpp
#ifnedef RESOURCES_LOAD
#define RESOURCES_LOAD
#include <fstream>
#include <sstream>
#include "MOGL/MOGL.hpp"

// Read the contents of a file ignoring any line that starts with a '#' 
// and put its contents in ss
void readFileContents(const std::string& filename, std::stringstream& ss)
{
    std::ifstream file;
    file.open(filename.c_str());
    std::string line;
    while (std::getline(file, line))
    {
        if (line[0] != '#')
        {
            ss << line;
            ss << '\n';
        }
    }
    file.close();
}

// Load an animation from a configuration file into the SpriteAnimationManager
// inside MultimediaOGL.
void loadAnimation(mogl::MultimediaOGL& mogl, const std::string& animation_name, const std::string& filename)
{
    std::ifstream animation_file;
    animation_file.open(filename.c_str());
    std::stringstream ss;
    readFileContents(filename, ss);
    mogl::SpriteAnimation animation;
    std::string texture_name;
    int num_frames;
    ss >> texture_name >> animation.offset_x >> animation.offset_y >> animation.margin_x >> animation.margin_y >> animation.width >> animation.height >> num_frames;
    animation.texture = mogl.textures().get(texture_name);
    int x;
    for (int i = 0; i < num_frames; ++i)
    {
```

9.6. Resources

```cpp
for (int i = 0; i < num_frames; ++i)
{
    ss >> x;
    animation.frame_order.push_back(x);
}
for (int i = 0; i < num_frames; ++i)
{
    ss >> x;
    animation.frame_time.push_back(hum::Time::milliseconds(x));
}
mogl.spriteAnimations().load(animation_name, animation);
```

// Load all resources defined in the resource definition file at <resources_filename>
void loadResources(mogl::MultimediaOGL& mogl, const std::string& resources_filename)
{
    std::stringstream ss;
    readFileContents(resources_filename, ss);
    std::string word, TYPE;
    while (ss >> word)
    {
        if (word == "TEXTURES" or word == "ANIMATIONS" or word == "SOUNDS" or word == "MUSIC")
        {
            TYPE = word;
            ss >> word;
        }
        if (TYPE == "TEXTURES")
        {
            std::string file;
            ss >> file;
            mogl.textures().load(word, "res/textures/" + file);
        }
        else if (TYPE == "ANIMATIONS")
        {
            std::string file;
            ss >> file;
            loadAnimation(mogl, word, "res/animations/" + file);
        }
        else if (TYPE == "SOUNDS")
        {
            std::string file;
            ss >> file;
            mogl.sounds().load(word, "res/sounds/" + file);
        }
        else if (TYPE == "MUSIC")
        {
            std::string file;
            ss >> file;
            mogl.music().load(word, "res/music/" + file);
        }
    }
} #endif
```
Math

A small file that implements square and includes cmath.

```cpp
# ifndef GAME_MATH_HPP
#define GAME_MATH_HPP

#include <cmath>

float square(float x) {
    return x*x;
}

# endif
```
10. Future work

The Hummingbird framework and MOGL, although functional, are far from being complete. The following list is a sample of what could be done to improve and extend the project.

- **Data locality**: Proper object pools for Actors and Behaviors to take advantage of caching and avoid the iteration over inactive Actors and Behaviors.

- **Hide the glm dependence from the user**: Use only glm classes internally, to the point that the eventual user of the MOGL plugin doesn’t need to know it is used inside (as in not having to include its headers).

- **3D rendering**: implement more Drawable Behaviors for rendering arbitrary 3D geometry.

There is a lot that can be done, in the end this is just the core of what a game engine is and inside every part of a game engine lots of tweaking can be done.
11. Project planning

Methodology and rigor

This project will be developed using the Agile\textsuperscript{12} method with sprints of one week. All tasks will be written as tickets in a Trello\textsuperscript{13} board. The board will consist of five lists: Backlog, To Do, In Progress, Done, Archive. The workflow with this lists is explained below.

First, all tasks created are put in the Backlog list. Once a week, every beginning of sprint iteration, the progress on the tickets assigned for the ending iteration is reviewed and the tickets in Done are moved to Archive.

Next, new tickets from Backlog are moved to To Do, bearing in mind the tickets that are still in To Do and In Progress (if any). The tickets put in To Do are the goal for the starting sprint and, ideally, will be done before the next one.

During the sprint, when picking a new ticket from To Do, it is moved to In Progress to make it clear that this is being worked on. This makes it easier to keep track of the progress during the sprint. When a ticket in progress is done, it is moved from In Progress to Done.

I chose this system because it allows for fast iteration on the implementation and enables me to adapt to changes or unexpected obstacles I may find. It also gives me feedback on my progress on-live as it is represented in the Trello board.

The described process will be approached, from a development point of view, using Git for version management and the feature branching\textsuperscript{1} technique for each ticket, merging each feature branch into master once the implementation has been tested to work properly. This workflow allows to simultaneously work on multiple tickets having each of them encapsulated in their respective branches and ensures that master branch never contains broken code.

Finally, to validate that the goals of this project have been achieved it’s only needed to refer to the scope exposed previously in this document and go through the list of features that must be implemented.
Description of tasks

In this section I’ll list and explain the tasks required to fulfill the project’s objective. These tasks are sorted by requirements, that is, every task requires the one before it to be able to start working on it. For all of the tasks a Laptop PC will be needed.

Project planning

This task consists mainly in all the tasks that the GEP course covers. It contains the following subtasks:

- Context and scope definition
- Temporal planning
- Budget and sustainability

This first task is very important, as it lays out the path for the whole project. It makes everything else almost straightforward as the overall view of the project is already thought.

The dependencies between the tasks are defined by the order they are presented. That means that the task 2 requires the task 1 to be done before being able to work on it, and so on with the following tasks.

Game framework

This is the main task for the project. It consist on the development, testing and documentation of all the parts that conform the game framework. This main task can be divided in the subtasks explained in the following sections.

Helper classes

This task consists of developing a set of classes containing general functionality for game development. These functionalities consist of describing spatial transformation of objects, measuring time and storing intervals of time. These functionalities will be encapsulated in the following classes, respectively: Transformation, Clock, Time.
11.2. Description of tasks

Actor and Behavior classes

This task consists of designing the behavior system and implementing the Behavior virtual class. It also includes implementing the Actor class, which will mainly be a container for behaviors, and its interface for being able to update the game state.

Game class

The Game class implements the main loop and contains the actor pool. At this stage this class won’t include the plug-in system, that will be implemented afterwards. Moreover, this task will also include extensive testing of all the system up to this point, looking for bugs and memory leaks.

Plug-in system

This task comes after the Game class is implemented and thoroughly tested. It consists in adding the plug-in system to the Game class. That is, designing and implementing the Plugin virtual class and extending the Game class to handle the plug-ins.

Once this task is done, the framework is completed and ready to use.

Multimedia OpenGL plug-in

This task consists in developing a Plugin for the game framework that wraps the SFML multimedia functionalities, in a way that is useful for developing 2D games with the framework and this plug-in, and a rendering pipeline using OpenGL.

Resource Managers

All games have resources that occupy space in memory such as textures or sounds among others. This task consists in developing managers for these resources to handle them efficiently.

Input handler

User input is what makes a game really a game, otherwise it’d be some kind of movie. This task will focus on implementing a data structure that stores the state of the input and allows querying that state in a simple way.
11.3. Time table

Drawable classes

This task consists on designing a Drawable interface for Behaviors that have a drawable representation. Then, implement the following drawable behaviors:

- *Rectangle*, a plain colored rectangle;
- *Circle*, a plain colored circle;
- *ConvexShape*, a plain colored convex polygon;
- *Text*, text using a font;
- *Sprite*, a static image from a texture;

All drawable behaviors will have their own transformation, relative to their actor.

MultimediaOGL class

The MultimediaOGL class will contain the logic for the rendering pipeline of the drawable behaviors, the input handles and the resource managers.

Again, at this point all the previous work will be extensively tested to find possible bugs and memory leaks.

Final task

This task is about making sure that all the documentation is done, reviewed to prepare the final presentation.

Time table

The following table summarizes the time that will be spent for each of the tasks described previously.
The planned schedule is represented in the following Gantt chart. I’ve considered a working
day of 4 hours a day for this schedule, as this will be approximately the time available to
dedicate to the project.
11.4. Gantt chart

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Start Date</th>
<th>End Date</th>
<th>Duration</th>
<th>Predecessors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project planning</td>
<td>02/22/16</td>
<td>03/15/16</td>
<td>17d</td>
<td></td>
</tr>
<tr>
<td>Game framework</td>
<td>03/16/16</td>
<td>04/08/16</td>
<td>18d</td>
<td>1</td>
</tr>
<tr>
<td>Helper classes</td>
<td>03/10/16</td>
<td>03/17/16</td>
<td>2d</td>
<td>1</td>
</tr>
<tr>
<td>Actor and Behavior classes</td>
<td>03/18/16</td>
<td>03/23/16</td>
<td>4d</td>
<td>3</td>
</tr>
<tr>
<td>Game class</td>
<td>03/24/16</td>
<td>03/31/16</td>
<td>6d</td>
<td>4</td>
</tr>
<tr>
<td>Plug-in system</td>
<td>04/01/16</td>
<td>04/08/16</td>
<td>6d</td>
<td>5</td>
</tr>
<tr>
<td>SFML Plug-in</td>
<td>04/11/16</td>
<td>05/02/16</td>
<td>16d</td>
<td>2</td>
</tr>
<tr>
<td>Resource managers</td>
<td>04/11/16</td>
<td>04/14/16</td>
<td>4d</td>
<td>2</td>
</tr>
<tr>
<td>Input handler</td>
<td>04/15/16</td>
<td>04/18/16</td>
<td>2d</td>
<td>8</td>
</tr>
<tr>
<td>Drawable classes</td>
<td>04/19/16</td>
<td>04/26/16</td>
<td>6d</td>
<td>9</td>
</tr>
<tr>
<td>SFMLPlugin class</td>
<td>04/27/16</td>
<td>05/02/16</td>
<td>4d</td>
<td>10</td>
</tr>
<tr>
<td>Final task</td>
<td>05/03/16</td>
<td>05/09/16</td>
<td>5d</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 11.2: Time planning of tasks

Figure 11.3: Gantt chart of the project
11.5. Action plan

The idea is to follow the plan as described above, completing tasks in the specified order and inside their planned time span. The plan described in the Gantt chart consists of a schedule of 4 hours a day and does not include weekends. If at some point there is some kind of obstacle that may slow the development of one of the tasks it is possible to work on weekends to get up-to-date with the planning.

Each task that consists of development also includes the documentation of said piece of code using Doxygen. This way, when the implementation is done I’ll immediately have its documentation without having to go back into the code.

Every time a major task is completed a meeting with the project director is to be scheduled to review the progress.

In the case of an unexpected extraordinary delay, the project planning will be adjusted accordingly by taking advantage of the time available between the planned ending of the project and the presentation date.

Budget estimation

For this section I am going to present an estimation of the budget needed to make the project possible. The costs will be divided in three groups depending on their type. These types are software, hardware and human resources.

To calculate the amortization I’ll consider the useful life of the resource and that the project will last 56 days, following the planning.

Hardware resources

The table below lists the hardware resources needed for the realization of the project, their cost and amortization.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Price</th>
<th>Useful life</th>
<th>Amortization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop PC</td>
<td>1000.00</td>
<td>4 years</td>
<td>38.35 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1000.00</td>
<td>-</td>
<td>38.35 €</td>
</tr>
</tbody>
</table>

*Figure 11.4: Hardware resources*
11.6. Budget estimation

Software resources

The next table lists the software resources needed for the project in the same format as the previous section.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Price</th>
<th>Useful life</th>
<th>Amortization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu 14.04 LTS</td>
<td>0.00 €</td>
<td>4 years</td>
<td>0.00 €</td>
</tr>
<tr>
<td>Vim</td>
<td>0.00 €</td>
<td>5 years</td>
<td>0.00 €</td>
</tr>
<tr>
<td>GCC 5.2.1</td>
<td>0.00 €</td>
<td>2 years</td>
<td>0.00 €</td>
</tr>
<tr>
<td>LibreOffice 5.0</td>
<td>0.00 €</td>
<td>5 years</td>
<td>0.00 €</td>
</tr>
<tr>
<td>SFML 2.3.2</td>
<td>0.00 €</td>
<td>1 year</td>
<td>0.00 €</td>
</tr>
<tr>
<td>Git</td>
<td>0.00 €</td>
<td>5 years</td>
<td>0.00 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0.00 €</strong></td>
<td><strong>-</strong></td>
<td><strong>0.00 €</strong></td>
</tr>
</tbody>
</table>

Figure 11.5: Software resources

Human resources

The table shows the costs for the human resources needed in the development of the project. The cost is calculated assuming a 4 hours per day schedule.

<table>
<thead>
<tr>
<th>Role</th>
<th>Price per hour</th>
<th>Time (days)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager</td>
<td>50.00 €</td>
<td>12</td>
<td>2400.00 €</td>
</tr>
<tr>
<td>Software architect</td>
<td>35.00 €</td>
<td>10</td>
<td>1400.00 €</td>
</tr>
<tr>
<td>Software developer</td>
<td>30.00 €</td>
<td>30</td>
<td>3600.00 €</td>
</tr>
<tr>
<td>Software tester</td>
<td>25.00 €</td>
<td>4</td>
<td>400.00 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>56</strong></td>
<td><strong>7800.00 €</strong></td>
</tr>
</tbody>
</table>

Figure 11.6: Human resources

Each of these different roles will be assigned different tasks in the project. Specifically, the Project manager will be assigned to almost the whole Project planning task and the Final task; the Software architect will work at the ending part of the Project planning determining the blueprints for the development of the project; the Software developer will take care of the whole development; and the Software tester will test the project when new milestones are reached during the development task.
11.7. Sustainability

Total budget

From the estimations in previous sections the following summary can be obtained, showing the total budget for the development of the project.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware resources</td>
<td>38.35 €</td>
</tr>
<tr>
<td>Software resources</td>
<td>0.00 €</td>
</tr>
<tr>
<td>Human resources</td>
<td>7800.00 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>7838.35 €</strong></td>
</tr>
</tbody>
</table>

*Figure 11.7: Total budget*

Budget control

The budget estimation as its name says is an estimation, therefore is not rigid and prone to changes.

The tasks that are more susceptible to deviations from the planning are the ones related to software architecture and development. If at any point there is a substantial delay in the development, the software developer will need to be hired for extra hours, thus changing the budget. On the other hand, I am confident with the planning and I don’t expect any big deviation from it.

Sustainability

In the following sections I’ll discuss the different areas of the sustainability of the project. Said areas are: economical sustainability, social sustainability and environmental sustainability.

Economical sustainability

In this document we have assessed the various costs of the resources required for the completion of the project.

The project is, in its core, a tool for reducing the work for a game developer. Once the framework is done, any future adjustments or updates will most probably be linked to a new
need for a feature of a future project. The framework has been designed to be easily extended so that the cost of future extensions should not be highly affected by the fact that they are extensions of the framework.

The cost of the project is relatively low and, although was not thought to be directly profitable, it will most probably be amortized when developing games with it, as it reduces the development time of video-games.

The service that this project will offer to its user could also be obtained from all the other game engine and frameworks that already exist in the market and open-source repositories. What this project offers is a lower level solution for those users that want to hand-craft their game from scratch.

Social sustainability

Nowadays, there is a high interest in game development. The video-game industry has grown a lot and many small studios have appeared. There have also appeared a set of full, free game-engines that have allowed for those small studios, and even students, to rapidly produce fully fleshed out products.

In this context, where these game-engines abstract all the inner systems from the user, some people may find they’ve lost control or that they want to learn the insides of a game. This is where this projects comes in. It offers the bare minimum systems for a game to be: a main loop, an actor pool with behaviors and a extension system to allow the user of the framework to extend it how they want, using any technology they want.

The scope of this project doesn’t just involve the main framework, but also a multimedia plug-in for it. Any future user of the product will see the implementation time of their game reduced by just needing to focus on the gameplay systems. Therefore, improving the life of the game developer using it.

Environmental sustainability

The resources required for this project are listed in the Budget estimation section of this document.

During the development of this project the only resource that will consume energy will be the Laptop PC. Assuming it is connected all the time to an electricity source and the charger consuming around 60W, through all the project it will consume approximately 13kWh, which is roughly equivalent to 3.9 kg of CO2. It is not a high amount of energy and may be reused by taking advantage of the laptop’s battery.
## Sustainability table

Considering the contents of this document, I’ve come up with the following table:

<table>
<thead>
<tr>
<th></th>
<th>Economic</th>
<th>Social</th>
<th>Environmental</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Planning</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assessment</td>
<td>Economic Viability</td>
<td>Improved quality of life</td>
<td>Resource analysis</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td><strong>Outcomes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assessment</td>
<td>Final cost versus</td>
<td>Impact on social</td>
<td>Resource</td>
</tr>
<tr>
<td></td>
<td>forecast</td>
<td>environment</td>
<td>consumption</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><strong>Risks</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assessment</td>
<td>Adapting to changes</td>
<td>Social damage</td>
<td>Environmental</td>
</tr>
<tr>
<td></td>
<td>of scenery</td>
<td></td>
<td>damage</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total assessment</strong></td>
<td>52</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 11.8: Sustainability table*
12. Conclusions

It has been a lot of work but I have learned more about the problems of implementing, not a game, but the scaffolding on top of which games are made.

Some features I’ve had to leave outside of the project due to time constraints and those were tough decisions I had to make in order to have a stable and functional product by the end.

Throughout the development, I managed to get feedback from some colleagues and my project’s director, Guillem Godoy, and I’ve taken their views in the implementation so that the framework was as simple to use as possible.

In the end, I am satisfied with the current status of the project. The objectives were accomplished and exceeded and I feel the architecture makes it easier for developers to make games and learn by extending the framework using any preferred technology; or just make a game using the MOGL plugin and focus on the game design and mechanics.
Bibliography


13. Appendix
Hummingbird Framework

Generated by Doxygen 1.8.11
## Contents

1 Hummingbird Framework .......................... 1  
1.1 Introduction .................................. 1

2 Namespace Index ................................ 3  
2.1 Namespace List ................................. 3

3 Hierarchical Index ............................... 5  
3.1 Class Hierarchy ................................ 5

4 Class Index ...................................... 7  
4.1 Class List ...................................... 7

5 Namespace Documentation ........................ 9  
5.1 hum Namespace Reference ..................... 9  
5.1.1 Detailed Description ....................... 10  
5.1.2 Function Documentation .................... 10  
5.1.2.1 assert_msg(bool condition, const TT &...tt) ........... 10  
5.1.2.2 log(const TT &...tt) ..................... 10  
5.1.2.3 log_d(const TT &...tt) ................... 11  
5.1.2.4 log_e(const TT &...tt) ................... 11

5.2 mogl Namespace Reference ..................... 11  
5.2.1 Detailed Description ....................... 12  
5.2.2 Class Documentation ....................... 13  
5.2.2.1 struct mogl::ShaderProgramDef ............. 13  
5.2.2.2 struct mogl::SpriteAnimation ............... 13  
5.2.3 Typedef Documentation ..................... 13  
5.2.3.1 SpaceTransformation .................... 14
6 Class Documentation

6.1 hum::Actor Class Reference .............................................. 15
6.1.1 Detailed Description .................................................. 16
6.1.2 Member Function Documentation .................................... 17
   6.1.2.1 addBehavior(Args &&...args) .................................... 17
   6.1.2.2 fixedUpdate() .................................................... 17
   6.1.2.3 game() .......................................................... 17
   6.1.2.4 game() const .................................................... 17
   6.1.2.5 getBehavior() ................................................... 17
   6.1.2.6 getBehavior() const .......................................... 18
   6.1.2.7 getBehaviors() .................................................. 18
   6.1.2.8 getBehaviors() const .......................................... 18
   6.1.2.9 isActive() ....................................................... 19
   6.1.2.10 onDestroy() ..................................................... 19
   6.1.2.11 preUpdate() .................................................... 19
   6.1.2.12 transform() .................................................... 19
   6.1.2.13 transform() const ............................................ 19

6.2 mogl::AnimatedSprite Class Reference ................................ 20
6.2.1 Detailed Description .................................................. 21
6.2.2 Constructor & Destructor Documentation ........................... 22
   6.2.2.1 AnimatedSprite(const SpriteAnimation &animation) ........ 22
6.2.3 Member Function Documentation .................................... 22
   6.2.3.1 draw() override ................................................. 22
   6.2.3.2 setSpriteAnimation(const SpriteAnimation &animation) ... 22

6.3 hum::Behavior Class Reference ........................................ 23
6.3.1 Detailed Description .................................................. 24
6.3.2 Member Function Documentation .................................... 24
   6.3.2.1 actor() .......................................................... 24
   6.3.2.2 actor() const ................................................... 24
   6.3.2.3 fixedUpdate() .................................................. 25
CONTENTS

6.3.2.4 init() ................................................. 25
6.3.2.5 onActivate() .............................................. 25

6.4 hum::exception::BehaviorNotFound Class Reference ........................................ 25
6.4.1 Detailed Description ........................................... 26

6.5 mogl::Camera Class Reference ............................................... 26
6.5.1 Detailed Description ........................................... 27
6.5.2 Constructor & Destructor Documentation ..................................... 27
6.5.2.1 Camera() ................................................ 27
6.5.3 Member Function Documentation ..................................... 27
6.5.3.1 setOrthogonal(float left, float right, float bottom, float top) ........... 27
6.5.3.2 setPerspective(float fovy, float aspect) .................................. 27

6.6 hum::Clock Class Reference ............................................... 28
6.6.1 Detailed Description ........................................... 28
6.6.2 Member Function Documentation ..................................... 28
6.6.2.1 getTime() const ..................................... 28
6.6.2.2 reset() ................................................ 28

6.7 mogl::Drawable Class Reference ............................................. 29
6.7.1 Detailed Description ........................................... 30
6.7.2 Member Function Documentation ..................................... 31
6.7.2.1 disable() ............................................. 31
6.7.2.2 draw()=0 ............................................. 31
6.7.2.3 enable() ............................................. 32
6.7.2.4 init() override ...................................... 32
6.7.2.5 isEnabled() const .................................. 32
6.7.2.6 onActivate() override .................................. 32
6.7.2.7 setShaderProgram(ShaderProgram *shader_program) .................. 32
6.7.2.8 transform() ........................................ 33
6.7.2.9 transform() const ................................... 33

6.8 hum::Game Class Reference .............................................. 33
6.8.1 Detailed Description ........................................... 34

Generated by Doxygen
<table>
<thead>
<tr>
<th>Section</th>
<th>Class Reference</th>
<th>Detailed Description</th>
<th>Member Function Documentation</th>
<th>Constructor &amp; Destructor Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8.2</td>
<td>Constructor &amp; Destructor Documentation</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>6.8.2.1</td>
<td>Game(unsigned int fixed_tickrate=60)</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>6.8.2.2</td>
<td>~Game()</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>6.8.3</td>
<td>Member Function Documentation</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>6.8.3.1</td>
<td>actors() const</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>6.8.3.2</td>
<td>addPlugin(Args &amp;&amp;...args)</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>6.8.3.3</td>
<td>deltaTime() const</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>6.8.3.4</td>
<td>fixedUpdateLag() const</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>6.8.3.5</td>
<td>fixedUpdateTime() const</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>6.8.3.6</td>
<td>getPlugin() const</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>6.8.3.7</td>
<td>getPlugin()</td>
<td>37</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>6.8.3.8</td>
<td>makeActor()</td>
<td>37</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>6.8.3.9</td>
<td>run()</td>
<td>37</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>6.9</td>
<td>mogl::InputHandler Class Reference</td>
<td>37</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6.9.1</td>
<td>Detailed Description</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6.9.2</td>
<td>Member Function Documentation</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6.9.2.1</td>
<td>handleEvent(const sf::Event &amp;event)</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6.9.2.2</td>
<td>update()</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6.10</td>
<td>hum::Kinematic Class Reference</td>
<td>39</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>6.10.1</td>
<td>Detailed Description</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>6.10.2</td>
<td>Member Function Documentation</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>6.10.2.1</td>
<td>init() override</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>6.11</td>
<td>hum::KinematicWorld Class Reference</td>
<td>41</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>6.11.1</td>
<td>Detailed Description</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>6.12</td>
<td>mogl::MultimediaOGL Class Reference</td>
<td>42</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>6.12.1</td>
<td>Detailed Description</td>
<td>44</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>6.12.2</td>
<td>Constructor &amp; Destructor Documentation</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>6.12.2.1</td>
<td>MultimediaOGL(sf::VideoMode mode, const sf::String &amp;title, sf::Uint32 style=sf::Style::Default, const sf::ContextSettings &amp;settings=sf::ContextSettings())</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>6.12.3</td>
<td>Member Function Documentation</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>6.12.3.1</td>
<td>setCamera(const Camera &amp;camera)</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
</tbody>
</table>
6.12.3.2 setDrawSpaceTransform(const SpaceTransformation &space_transform) . . . 46

6.13 mogl::MusicManager Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.13.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.13.2 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.13.2.1 loadResource(const std::string &name) override . . . . . . . . . . . . . . . . 47

6.14 hum::Plugin Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
6.14.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
6.14.2 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
6.14.2.1 game() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
6.14.2.2 game() const . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

6.15 hum::exception::PluginNotFound Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.15.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

6.16 mogl::Rectangle Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.16.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
6.16.2 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
6.16.2.1 init() override . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
6.16.2.2 setShaderProgram(ShaderProgram *shader_program) override . . . . . . . . . . . 52

6.17 hum::ResourceManager<T, Key, Data> Class Template Reference . . . . . . . . . . . . . . . . . . . 52
6.17.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
6.17.2 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
6.17.2.1 load(const Key &key, const Data &data) . . . . . . . . . . . . . . . . . . . . . . . . 53
6.17.2.2 loadResource(const Data &data)=0 . . . . . . . . . . . . . . . . . . . . . . . . . 54

6.18 mogl::Shader Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
6.18.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
6.18.2 Member Function Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.18.2.1 isCompiled() const . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.18.2.2 loadFromFile(const Type type, const std::string &filename) . . . . . . . . . . . . . . . . . 55
6.18.2.3 loadFromSource(const Type type, const std::string &source) . . . . . . . . . . . . . . . . . . . 55

6.19 mogl::ShaderProgram Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.19.1 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
6.19.2 Member Function Documentation

6.19.2.1 addShader(const Shader &shader) ............................................. 57
6.19.2.2 bindFragmentOutput(const std::string &output_name) .................... 57
6.19.2.3 bindVertexAttribute(const std::string &attrib_name, GLint size, GLsizei stride, void *first_pointer) ......................................... 57
6.19.2.4 isLinked() ........................................................................ 58
6.19.2.5 link() ........................................................................... 58
6.19.2.6 setUniform2f(const std::string &uniform_name, float v0, float v1) .... 58
6.19.2.7 setUniform3f(const std::string &uniform_name, float v0, float v1, float v2) ................................................................. 58
6.19.2.8 setUniform4f(const std::string &uniform_name, float v0, float v1, float v2, float v3) ................................................................. 58
6.19.2.9 setUniformMatrix4f(const std::string &uniform_name, const glm::mat4 &mat) ................................................................. 59
6.19.2.10 use() ........................................................................... 59

6.20 mogl::ShaderProgramManager Class Reference ................................................. 59
6.20.1 Detailed Description ........................................................................ 60

6.21 mogl::SoundManager Class Reference ....................................................... 61
6.21.1 Detailed Description ........................................................................ 62
6.21.2 Member Function Documentation ....................................................... 62
6.21.2.1 clearSounds() ................................................................... 62
6.21.2.2 get(SoundId sound_id) ......................................................... 63
6.21.2.3 get(SoundId sound_id) const .................................................. 63
6.21.2.4 loadResource(const std::string &name) override ......................... 63
6.21.2.5 play(const std::string &name, int volume, bool loop=false, bool relative_to_listener=false, const sf::Vector3f &position=sf::Vector3f()) ........ 63

6.22 mogl::Sprite Class Reference ................................................................. 64
6.22.1 Detailed Description ........................................................................ 65
6.22.2 Constructor & Destructor Documentation ............................................. 66
6.22.2.1 Sprite(const sf::Texture *texture, const sf::IntRect &rect=sf::IntRect()) ........ 66
6.22.3 Member Function Documentation ....................................................... 66
6.22.3.1 draw() override .................................................................. 66
6.22.3.2 init() override ................................................................... 66
6.22.3.3 setShaderProgram(ShaderProgram *shader_program) override ....... 66
6.27.4.3 operator ∗(T left, const hum::Vector2< T >&right) ........................................ 76
6.27.4.4 operator+=(hum::Vector2< T >&left, T right) .................................................. 77
6.27.4.5 operator+(const hum::Vector2< T >&left, const hum::Vector2< T >&right) .. 77
6.27.4.6 operator+=(hum::Vector2< T >&left, const hum::Vector2< T >&right) .......... 77
6.27.4.7 operator-(const hum::Vector2< T >&right) ...................................................... 78
6.27.4.8 operator-(const hum::Vector2< T >&left, const hum::Vector2< T >&right) .. 78
6.27.4.9 operator-=(hum::Vector2< T >&left, const hum::Vector2< T >&right) .......... 78
6.27.4.10 operator/(const hum::Vector2< T >&left, T right) ......................................... 79
6.27.4.11 operator/=(hum::Vector2< T >&left, T right) ............................................... 79
6.27.4.12 operator==(const hum::Vector2< T >&left, const hum::Vector2< T >&right) .. 79

6.28 hum::Vector3< T > Class Template Reference ....................................................... 80
6.28.1 Detailed Description ......................................................................................... 81
6.28.2 Constructor & Destructor Documentation ......................................................... 82
6.28.2.1 Vector3(const Vector3< U >&v) ................................................................. 82
6.28.3 Member Function Documentation .................................................................. 82
6.28.3.1 operator[](unsigned int position) ............................................................... 82
6.28.3.2 operator[](unsigned int position) const ...................................................... 82
6.28.4 Friends And Related Function Documentation ............................................... 82
6.28.4.1 operator"!=(const hum::Vector3< T >&left, const hum::Vector3< T >&right) .. 82
6.28.4.2 operator=(const hum::Vector3< T >&left, T right) ......................................... 83
6.28.4.3 operator=(T left, const hum::Vector3< T >&right) ......................................... 83
6.28.4.4 operator-=(hum::Vector3< T >&left, T right) .................................................. 83
6.28.4.5 operator-=(const hum::Vector3< T >&left, const hum::Vector3< T >&right) .. 83
6.28.4.6 operator-=(hum::Vector3< T >&left, const hum::Vector3< T >&right) .......... 84
6.28.4.7 operator-(const hum::Vector3< T >&right) ...................................................... 84
6.28.4.8 operator-(const hum::Vector3< T >&left, const hum::Vector3< T >&right) .. 84
6.28.4.9 operator-=(hum::Vector3< T >&left, const hum::Vector3< T >&right) .......... 85
6.28.4.10 operator/(const hum::Vector3< T >&left, T right) ......................................... 85
6.28.4.11 operator/=(hum::Vector3< T >&left, T right) ............................................... 85
6.28.4.12 operator==(const hum::Vector3< T >&left, const hum::Vector3< T >&right) .. 86

Index .......................................................................................................................... 87
Chapter 1

Hummingbird Framework

1.1 Introduction

C++ game development framework.
Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

- **hum**: Namespace for the Hummingbird framework .......................... 9
- **mogl**: Namespace for the Multimedia OpenGL (MOGL) Plugin ................ 11
Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

hum::Actor .......................................................... 15
hum::Behavior ....................................................... 23
hum::Behavior ....................................................... 23
mogl::Drawable ..................................................... 29
mogl::Drawable ..................................................... 29
mogl::Rectangle .................................................... 50
mogl::Sprite ....................................................... 64
mogl::AnimatedSprite ........................................... 20
mogl::Camera ....................................................... 26
hum::Clock .......................................................... 28
std::exception
  hum::exception::BehaviorNotFound ................................ 25
  hum::exception::PluginNotFound ................................ 50
hum::Game .......................................................... 33
mogl::InputHandler .............................................. 37
hum::Plugin ........................................................ 48
hum::KinematicWorld ............................................. 41
mogl::MultimediaOGL ............................................ 42
hum::ResourceManager< T, Key, Data > .......................... 52
hum::ResourceManager< sf::Music > .............................. 52
mogl::MusicManager .............................................. 46
hum::ResourceManager< sf::SoundBuffer > ..................... 52
mogl::SoundManager .............................................. 61
hum::ResourceManager< sf::Texture > .......................... 52
mogl::TextureManager ............................................ 68
hum::ResourceManager< ShaderProgram, std::string, ShaderProgramDef > .......................... 52
mogl::ShaderProgramManager .................................. 59
hum::ResourceManager< SpriteAnimation, std::string, SpriteAnimation > ..................... 52
mogl::SpriteAnimationManager ................................ 67
mogl::Shader ....................................................... 54
mogl::ShaderProgram ............................................ 55
mogl::ShaderProgramDef .......................................... 11
mogl::SpriteAnimation ........................................... 11
hum::Time ........................................................... 70
hum::Transformation ............................................. 71
hum::Vector2< T > .................................................. 73
hum::Vector3< T > .................................................. 80
Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

hum::Actor
Actor (also known as GameObject) .............................................. 15

mogl::AnimatedSprite
A textured rectangle-shaped 1x1 Drawable that changes over time ............... 20

hum::Behavior
Class from which inherit to implement and give an Actor behavior ................. 23

hum::exception::BehaviorNotFound
Exception thrown when getting a Behavior from an Actor that does not contain it. (see Actor::getBehavior()) ............................................. 25

mogl::Camera
The Camera is the device through which the player views the world ................. 26

hum::Clock
Class for measuring intervals of time ........................................... 28

mogl::Drawable
Abstract class to implement a hum::Behavior that can be drawn .................... 29

hum::Game
The class that runs the Game ..................................................... 33

mogl::InputHandler
Class that handles SFML input events and allows for easy quering of related data .... 37

hum::Kinematic
Add Kinematic properties to the Actor. (Requires KinematicWorld) ............... 39

hum::KinematicWorld
Plugin that handles the transformation (movement, scale or rotation) of an Actor with a Kinematic behavior ................................................. 41

mogl::MultimediaOGL
Plugin that handles all the rendering pipeline, input and resource manangement .... 42

mogl::MusicManager
A hum::ResourceManager for sf::Music ......................................... 46

hum::Plugin
Class from which inherit to implement and give a Plugin for the Game ............ 48

hum::exception::PluginNotFound
Exception thrown when getting a Plugin from a Game that does not contain it. (see Game::getPlugin()) .................................................. 50

mogl::Rectangle
A rectangle-shaped 1x1 Drawable ................................................ 50
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hum::ResourceManager&lt; T, Key, Data &gt;</code></td>
<td>Class that implements the generic functionality of a resource manager</td>
<td>52</td>
</tr>
<tr>
<td><code>mogl::Shader</code></td>
<td>Class for loading a Vertex Shader or a Fragment Shader</td>
<td>54</td>
</tr>
<tr>
<td><code>mogl::ShaderProgram</code></td>
<td>Class to load, handle and work with shader programs</td>
<td>55</td>
</tr>
<tr>
<td><code>mogl::ShaderProgramManager</code></td>
<td>A <code>hum::ResourceManager</code> for <code>mogl::ShaderProgram</code></td>
<td>59</td>
</tr>
<tr>
<td><code>mogl::SoundManager</code></td>
<td>A <code>hum::ResourceManager</code> for <code>sf::SoundBuffers</code></td>
<td>61</td>
</tr>
<tr>
<td><code>mogl::Sprite</code></td>
<td>A textured rectangle-shaped 1x1 <code>Drawable</code></td>
<td>64</td>
</tr>
<tr>
<td><code>mogl::SpriteAnimationManager</code></td>
<td>A <code>hum::ResourceManager</code> for <code>mogl::SpriteAnimations</code></td>
<td>67</td>
</tr>
<tr>
<td><code>mogl::TextureManager</code></td>
<td>A <code>hum::ResourceManager</code> for <code>sf::Texture</code></td>
<td>68</td>
</tr>
<tr>
<td><code>hum::Time</code></td>
<td>Class for representing intervals of time. This class has nanoseconds precision</td>
<td>70</td>
</tr>
<tr>
<td><code>hum::Transformation</code></td>
<td>Class representing 3D spatial transformation of an object</td>
<td>71</td>
</tr>
<tr>
<td><code>hum::Vector2&lt; T &gt;</code></td>
<td>Class representing a 2D vector</td>
<td>73</td>
</tr>
<tr>
<td><code>hum::Vector3&lt; T &gt;</code></td>
<td>Class representing a 3D vector</td>
<td>80</td>
</tr>
</tbody>
</table>
Chapter 5

Namespace Documentation

5.1 hum Namespace Reference

Namespace for the Hummingbird framework.

Classes

- class Actor
  
  Actor (also known as GameObject).

- class Behavior
  
  Class from which inherit to implement and give an Actor behavior.

- class Clock
  
  Class for measuring intervals of time.

- class Game
  
  The class that runs the Game.

- class Kinematic
  
  Add Kinematic properties to the Actor. (Requires KinematicWorld).

- class KinematicWorld
  
  Plugin that handles the transformation (movement, scale or rotation) of an Actor with a Kinematic behavior.

- class Plugin
  
  Class from which inherit to implement and give a Plugin for the Game.

- class ResourceManager
  
  Class that implements the generic functionality of a resource manager.

- class Time
  
  Class for representing intervals of time. This class has nanoseconds precision.

- class Transformation
  
  Class representing 3D spacial transformation of an object.

- class Vector2
  
  Class representing a 2D vector.

- class Vector3
  
  Class representing a 3D vector.
Functions

• template<typename... TT>
  void assert_msg (bool condition, const TT &...tt) [inline]
  
  Check a condition and if it fails, exit the program and print the message.

• template<typename... TT>
  void log (const TT &...tt) [inline]
  
  Print a message to the standard output.

• template<typename... TT>
  void log_e (const TT &...tt) [inline]
  
  Print a message to the error output.

• template<typename... TT>
  void log_d (const TT &...tt) [inline]
  
  Print a message to the standard output.

5.1.1 Detailed Description

Namespace for the Hummingbird framework.

5.1.2 Function Documentation

5.1.2.1 template<typename... TT> void hum::assert_msg ( bool condition, const TT &... tt ) [inline]

Check a condition and if it fails, exit the program and print the message.

This method does nothing if NDEBUG is defined.

Usage example:

```
hum::assert_msg(player_x > 64, "Player is outside of the map! x=", player_x);
```

5.1.2.2 template<typename... TT> void hum::log ( const TT &... tt ) [inline]

Print a message to the standard output.

T can be any type that has the operator << overloaded. It can also be any of the following classes:

• hum::Vector2
• hum::Vector3
• hum::Transformation
• hum::Time
• hum::Clock

Usage example:

```
hum::log("Player position: ", actor().transform().position);
```
5.1.2.3 template<typename... TT> void hum::log_d ( const TT &... tt ) [inline]

Print a message to the standard output.

This method does nothing if NDEBUG is defined.

T can be any type that has the operator << overloaded. It can also be any of the following classes:

- hum::Vector2
- hum::Vector3
- hum::Transformation
- hum::Time
- hum::Clock

Usage example:

```cpp
hum::log_d("Player position: ", actor().transform().position);
```

5.1.2.4 template<typename... TT> void hum::log_e ( const TT &... tt ) [inline]

Print a message to the error output.

T can be any type that has the operator << overloaded. It can also be any of the following classes:

- hum::Vector2
- hum::Vector3
- hum::Transformation
- hum::Time
- hum::Clock

Usage example:

```cpp
hum::log_e("Player position: ", actor().transform().position);
```
Classes

- **class AnimatedSprite**
  A textured rectangle-shaped 1x1 Drawable that changes over time.

- **class Camera**
  The Camera is the device through which the player views the world.

- **class Drawable**
  Abstract class to implement a hum::Behavior that can be drawn.

- **class InputHandler**
  Class that handles SFML input events and allows for easy querying of related data.

- **class MultimediaOGL**
  Plugin that handles all the rendering pipeline, input and resource management.

- **class MusicManager**
  A hum::ResourceManager for sf::Music.

- **class Rectangle**
  A rectangle-shaped 1x1 Drawable.

- **class Shader**
  Class for loading a Vertex Shader or a Fragment Shader.

- **class ShaderProgram**
  Class to load, handle and work with shader programs.

- **struct ShaderProgramDef**
  A ShaderProgram definition to use with ShaderProgramManager. More...

- **class ShaderProgramManager**
  A hum::ResourceManager for mogl::ShaderProgram.

- **class SoundManager**
  A hum::ResourceManager for sf::SoundBuffers.

- **class Sprite**
  A textured rectangle-shaped 1x1 Drawable.

- **struct SpriteAnimation**
  A description of the animation to be played by a AnimatedSprite. More...

- **class SpriteAnimationManager**
  A hum::ResourceManager for mogl::SpriteAnimations.

- **class TextureManager**
  A hum::ResourceManager for sf::Texture.

Typedefs

- **typedef std::function<void(const hum::Game &, hum::Transformation &)> SpaceTransformation**
  Signature for a space transformation method. (see MultimediaOGL::setDrawSpaceTransform()).

- **typedef unsigned int SoundId**

- **typedef std::pair<SoundId, sf::Sound*> SoundPair**

Functions

- **void defaultSpaceTransform (const hum::Game &, hum::Transformation &)**

5.2.1 Detailed Description

Namespace for the Multimedia OpenGL (MOGL) Plugin.
5.2.2 Class Documentation

5.2.2.1 struct mogl::ShaderProgramDef

A ShaderProgram definition to use with ShaderProgramManager.

Collaboration diagram for mogl::ShaderProgramDef:

```
A ShaderProgram definition to use with ShaderProgramManager.

Class Members

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>fragment_output_name</td>
<td>Name of the output of the fragment shader.</td>
</tr>
<tr>
<td>Shader &amp;</td>
<td>fragment_shader</td>
<td>Fragment Shader.</td>
</tr>
<tr>
<td>Shader &amp;</td>
<td>vertex_shader</td>
<td>Vertex Shader.</td>
</tr>
</tbody>
</table>
```

5.2.2.2 struct mogl::SpriteAnimation

A description of the animation to be played by a AnimatedSprite.

Class Members

```
vector< unsigned int > | frame_order       | Sequence of ids of the tiles in the tilesheet to play. |
vector< Time >          | frame_time        | Sequence of hum::Times for each frame in the animation. |
int                      | height            | Height of a tile in the tilesheet.                 |
int                      | margin_x          | Horizontal margin between tiles in the tilesheet.  |
int                      | margin_y          | Vertical margin between tiles in the tilesheet.    |
int                      | offset_x          | Horizontal offset of the tilesheet.               |
int                      | offset_y          | Vertical offset of the tilesheet.                 |
Texture *                | texture           | Texture to be used.                               |
int                      | width             | Width of a tile in the tilesheet.                 |
```

5.2.3 Typedef Documentation

Generated by Doxygen
5.2.3.1 typedef std::function<void(const hum::Game&, hum::Transformation&)> mogl::SpaceTransformation

Signature for a space transformation method. (see MultimediaOGL::setDrawSpaceTransform()).

The first parameter is the instance of the hum::Game and the second is the hum::Transformation of the Drawable to be drawn, not the one relative to its hum::Actor, but the accumulated one.
Chapter 6

Class Documentation

6.1 hum::Actor Class Reference

Actor (also known as GameObject).

#include <Actor.hpp>

Public Member Functions

• ~Actor ()
  Class destructor.
• void preUpdate ()
  Pre update.
• void fixedUpdate ()
  Fixed update.
• void onDestroy ()
  Called just before destroying the Actor.
• Game & game ()
  Get the Actor's Game.
• const Game & game () const
  Get the Actor's Game.
• Transformation & transform ()
  Get the Actor's Transformation.
• const Transformation & transform () const
  Get the Actor's Transformation.
• bool isActive ()
  Check if the Actor is active.
• void activate ()
  Set the Actor active.
• void deactivate ()
  Set the Actor inactive.
• template<typename B , class... Args>
  B * addBehavior (Args &&...args)
  Add a new Behavior to the Actor.
6.1.1 Detailed Description

**Actor** (also known as GameObject).

This class represents an object in the Game. You can create a new **Actor** by calling **Game::makeActor()**. This method will create a new **Actor** and return it. The **Game** owns the **Actor** and it controls its lifetime.

To destroy an **Actor** you **must** call **Game::destroy()**, not its destructor. The **Actor** then, will be marked to be destroyed and after the next update step it’ll be deleted. Just before being deleted, the **Actor** will call its Behaviors **onDestroy()** method.

An **Actor**, by default, doesn’t have any behaviour and you **should not inherit from** it. Instead **Actors** are composed by **Behaviors**. These **Behaviors** are the ones that must implement the behaviour of the **Actor** composed by them.

On the other hand, **Actors** do have a Transformation, accessible through **transform()** and a reference to its **Game**, accessible through **game()**.

**A Actor** can be **active** or **inactive**. If a **Actor** is **inactive** it exists, all its **Behaviors** also exist and have been instantiated; but it **won’t** be updated. Same applies to its **Behaviors**.

Usage example. The **Actor** will be destroyed after 10 fixed updates:

```cpp
// We define two Behaviors: A and B.
class B : public hum::Behavior
{
public:
  B(int x): value(x)
  {}
  int value;
};

class A : public hum::Behavior
{
public:
  A(int x): current(x)
  {}
  void init() override
  {
    last = actor().getBehavior<B>()->value;
  }
  void fixedUpdate() override
  {
    current++;
    if (current > last)
      { actor().game().destroy(actor());
    }
    hum::log("Count: ", current);
  }
};
```
6.1.2 Member Function Documentation

6.1.2.1 template<typename B, class... Args> B* hum::Actor::addBehavior ( Args&&... args ) [inline]

Add a new Behavior to the Actor.

Template Parameters

<table>
<thead>
<tr>
<th>B</th>
<th>The Behavior's type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Args</td>
<td>The parameters for B's constructor.</td>
</tr>
</tbody>
</table>

This method will create a new instance of the given type B, add it to the Actor.

Returns

A pointer to the new instance of the given Behavior type.

6.1.2.2 void hum::Actor::fixedUpdate ( )

Fixed update.

This method is called every Game's fixed update. It call all its Behaviors' fixedUpdate().

6.1.2.3 Game& hum::Actor::game ( )

Get the Actor's Game.

Returns

The Game this Actor belongs.

6.1.2.4 const Game& hum::Actor::game ( ) const

Get the Actor's Game.

Returns

The Game this Actor belongs.

6.1.2.5 template< typename T > T* hum::Actor::getBehavior ( ) throw ( exception::BehaviorNotFound ) [inline]

Get a Behavior of the given type in the Actor.
Template Parameters

\texttt{T} The Behavior's type.

if the Actor doesn't have a Behavior of type \texttt{T} it will throw a exception::BehaviorNotFound.

Returns

A pointer to the first Behavior of type \texttt{T} in the Actor.

6.1.2.6 template<typename \texttt{T} > const \texttt{T} * hum::Actor::getBehavior ( ) const throw exception::BehaviorNotFound) [inline]

Get a Behavior of the given type in the Actor.

Template Parameters

\texttt{T} The Behavior's type.

if the Actor doesn't have a Behavior of type \texttt{T} it will throw a exception::BehaviorNotFound.

Returns

A pointer to the first Behavior of type \texttt{T} in the Actor.

6.1.2.7 template<typename \texttt{T} > std::vector<\texttt{T} *> hum::Actor::getBehaviors ( ) [inline]

Get all Behaviors of a type in the Actor.

Template Parameters

\texttt{T} The Behavior's type.

Returns

A std::vector<\texttt{T} *> with all the Behaviors of type \texttt{T}, if any.

6.1.2.8 template<typename \texttt{T} > std::vector<const \texttt{T} *> hum::Actor::getBehaviors ( ) const [inline]

Get all Behaviors of a type in the Actor.

Template Parameters

\texttt{T} The Behavior's type.
Returns

A std::vector<T*> with all the Behaviors of type T, if any.

6.1.2.9 bool hum::Actor::isActive ()

Check if the Actor is active.
If an Actor is not active, it is not updated

Returns

true if the Actor is active, false otherwise.

6.1.2.10 void hum::Actor::onDestroy ()

Called just before destroying the Actor.
It calls its Behaviors' onDestroy() method.

6.1.2.11 void hum::Actor::preUpdate ()

Pre update.
In this method are initialized the actor's Behaviors that haven't been initialized up to this point.

6.1.2.12 Transformation& hum::Actor::transform ()

Get the Actor's Transformation.

Returns

This Actor's Transformation.

6.1.2.13 const Transformation& hum::Actor::transform () const

Get the Actor's Transformation.

Returns

This Actor's Transformation.

The documentation for this class was generated from the following file:

• hummingbird/include/hummingbird/Actor.hpp

Generated by Doxygen
6.2 mogl::AnimatedSprite Class Reference

A textured rectangle-shaped 1x1 Drawable that changes over time.

```
#include <AnimatedSprite.hpp>
```

Inheritance diagram for mogl::AnimatedSprite:

```
hum::Behavior

mogl::Drawable

mogl::Sprite

mogl::AnimatedSprite
```

Collaboration diagram for mogl::AnimatedSprite:

```
hum::Behavior

mogl::Drawable

mogl::Sprite

mogl::AnimatedSprite
```
Public Types

- enum Status { PLAYING, PAUSED, STOPPED }
  
  The various statuses in which an AnimatedSprite can be regarding the playback of an animation.

Public Member Functions

- AnimatedSprite (const SpriteAnimation *animation)
  
  Class constructor with SpriteAnimation to use.

- const SpriteAnimation *spriteAnimation () const
  
  Get the SpriteAnimation being used.

- void setSpriteAnimation (const SpriteAnimation *animation)
  
  Get the SpriteAnimation being used.

- void play ()
  
  Start or continue the animation's playback.

- void pause ()
  
  Pause the animation's playback.

- void stop ()
  
  Stop the animation's playback.

- Status status () const
  
  Get the status of the playback.

- unsigned int frameIndex () const
  
  Get the current frame in the animation.

- void frameIndex (unsigned int frame_index)
  
  Set the current frame in the animation.

- void setLooping (bool looping)
  
  Set whether the animation should loop or not.

- bool isLooping () const
  
  Get whether the animation is loop or not.

- void draw () override
  
  Draw the Drawable to the active OpenGL context.

Static Public Member Functions

- static const char *behaviorName ()

Additional Inherited Members

6.2.1 Detailed Description

A textured rectangle-shaped 1x1 Drawable that changes over time.

For different sizes use the scale in either the hum::Actor hum::Transform of the Drawable's hum::Transform.
6.2.2 Constructor & Destructor Documentation

6.2.2.1 mogl::AnimatedSprite::AnimatedSprite ( const SpriteAnimation ∗ animation )

Class constructor with SpriteAnimation to use.

The AnimatedSprite doesn't handle the given SpriteAnimation pointer and the pointed animation must exists while the AnimatedSprite is using it.

By default, an AnimatedSprite is playing and looping.

6.2.3 Member Function Documentation

6.2.3.1 void mogl::AnimatedSprite::draw ( ) [override],[virtual]

Draw the Drawable to the active OpenGL context.

This abstract method is to be implemented by derived classes to define how the Drawable must be drawn.

By the point where this method is called, the ShaderProgram of the Drawable is being used and the model, view and projection uniform matrices are set.

Implements mogl::Drawable.

6.2.3.2 void mogl::AnimatedSprite::setSpriteAnimation ( const SpriteAnimation ∗ animation )

Get the SpriteAnimation being used.

The AnimatedSprite doesn't handle the given SpriteAnimation pointer and the pointed animation must exists while the AnimatedSprite is using it.

The documentation for this class was generated from the following file:

• MOGL/include/MOGL/AnimatedSprite.hpp
6.3  

6.3 hum::Behavior Class Reference

Class from which inherit to implement and give an Actor behavior.

#include <Behavior.hpp>

Inheritance diagram for hum::Behavior:

```
hum::Behavior
  ↓
hum::Kinematic  mogl::Drawable
  ↓
mogl::Rectangle  mogl::Sprite
  ↓
mogl::AnimatedSprite
```

Public Member Functions

- virtual ~Behavior ()
  
  Class destructor.
- virtual void init ()
  
  Behavior initialization function.
- virtual void fixedUpdate ()
  
  Fixed update.
- virtual void onActivate ()
  
  Called when the Actor is activated.
- virtual void onDeactivate ()
  
  Called when the Actor is deactivated.
- virtual void onDestroy ()
  
  Called when the Actor is about to be destroyed.
- Actor & actor ()
  
  Get the Behavior's Actor.
- const Actor & actor () const
  
  Get the Behavior's Actor.
6.3.1 Detailed Description

Class from which inherit to implement and give an Actor behavior.

A Behavior always lives inside an Actor. Said actor takes care of the lifecycle of the Behavior.

For creating a custom Behavior you may inherit from this class and override the methods you need to implement the wanted functionality.

Behaviors must also implement a static const char* behaviorName() method that, as the name hints, returns the class name. This is used for error reporting.

Usage example:

class PrintTransform : public hum::Behavior
{
public:
  void init() override
  {
    hum::log("Behavior initialized");
  }

  void fixedUpdate() override
  {
    hum::log("Actor transformation: ", actor().transform());
  }

  void onDestroy() override
  {
    hum::log("Actor destroyed");
  }

  static const char* behaviorName()
  {
    return "PrintTransform";
  }
};

6.3.2 Member Function Documentation

6.3.2.1 Actor& hum::Behavior::actor ( )

Get the Behavior's Actor.

Returns
  The Behavior's Actor

6.3.2.2 const Actor& hum::Behavior::actor ( ) const

Get the Behavior's Actor.

Returns
  The Behavior's Actor
6.3.2.3 virtual void hum::Behavior::fixedUpdate() [virtual]

Fixed update.

The main method for updating the logic of the game.

This method is called every fixed update. The frequency of the fixed update depends on the settings of the Game that contains this Behavior's Actor.

6.3.2.4 virtual void hum::Behavior::init() [virtual]

Behavior initialization function.

This method is called only once, the first preUpdate of the behavior in the Actor.

This method is useful for initializing values that require accessing other Behaviors in the Actor, as it guarantees that they are accessible from this point on.

It also guarantees that the Game is running and all Plugins' gameStart() have been called at this point, therefore the Game has been initialized.

Reimplemented in mogl::Drawable, mogl::Sprite, hum::Kinematic, and mogl::Rectangle.

6.3.2.5 virtual void hum::Behavior::onActivate() [virtual]

Called when the Actor is activated.

This method is not called when the Behavior is initialized as it is assumed that a Behavior is only initialized if its Actor is active.

Reimplemented in mogl::Drawable.

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Behavior.hpp

6.4 hum::exception::BehaviorNotFound Class Reference

Exception thrown when getting a Behavior from an Actor that does not contain it. (see Actor::getBehavior())

#include <Exceptions.hpp>

Inherits std::exception.

Public Member Functions

- BehaviorNotFound (const char *type_name)
- virtual const char * what() const noexcept
6.4.1 Detailed Description

Exception thrown when getting a Behavior from an Actor that does not contain it. (see Actor::getBehavior())

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Exceptions.hpp

6.5 mogl::Camera Class Reference

The Camera is the device through which the player views the world.

#include <Camera.hpp>

Public Member Functions

- Camera ()
  
  Default constructor.

- void setPerspective (float fovy, float aspect)
  
  Set the Camera to perspective projection.

- void setOrthogonal (float left, float right, float bottom, float top)
  
  Set the Camera to orthogonal projection.

- void setPosition (const hum::Vector3f &position)
  
  Set the position of the camera.

- const hum::Vector3f & getPosition () const
  
  Get the position of the camera.

- void setCenter (const hum::Vector3f &center)
  
  Set the center of the camera. (Where it is looking at).

- const hum::Vector3f & getCenter () const
  
  Get the center of the camera. (Where it is looking at).

- void setUp (const hum::Vector3f &up)
  
  Set the up vector of the camera.

- const hum::Vector3f & getUp () const
  
  Get the up vector of the camera.

- void setZNear (float z_near)
  
  Set the Z near of the camera.

- float getZNear () const
  
  Get the Z near of the camera.

- void setZFar (float z_far)
  
  Set the Z far of the camera.

- float getZFar () const
  
  Get the Z far of the camera.

- const glm::mat4 & getProjection ()
  
  Get the projection matrix of the camera. (Internal use only).

- const glm::mat4 & getView ()
  
  Get the view matrix of the camera. (Internal use only).

- bool projectionChanged () const
  
  Return whether the projection matrix has changed since the last getProjection(). (Internal use only).

- bool viewChanged () const
  
  Return whether the view matrix has changed since the last getView(). (Internal use only).
6.5 mogl::Camera Class Reference

6.5.1 Detailed Description

The Camera is the device through which the player views the world.

It is used by MultimediaOGL to render the game world.

By default the camera is set to be orthogonal. It is placed at the point (0, 0, -1) and looks towards (0, 0, 1) with a
viewport of 100 by 100. The (0, 0) is located at the top left corner with the x-axis growing to the right and the y-axis
growing downwards.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 mogl::Camera::Camera ( )

Default constructor.

Constructs the camera with the following default values:

```cpp
mogl::Camera cam;
cam.setZNear(0.1f);
cam.setZFar(1000.f);
cam.setOrthogonal(0, -100, 100, 0);
cam.setPosition(hum::Vector3f(0, 0, -1));
cam.setCenter(hum::Vector3f(0, 0, 1));
cam.setUp(hum::Vector3f(0, -1, 0));
```

6.5.3 Member Function Documentation

6.5.3.1 void mogl::Camera::setOrthogonal ( float left, float right, float bottom, float top )

Set the Camera to orthogonal projection.

The rectangle of the view is formed by the points:

```
+----------------------------------------+
|                                   |
|                                   |
|                                   |
|                                   |
+----------------------------------------+
```

Where position is the position of the camera and the normal of the plane is (center - position), where center is
the point at which the camera is looking at.

6.5.3.2 void mogl::Camera::setPerspective ( float fovy, float aspect )

Set the Camera to perspective projection.

The perspective is defined by the given vertical field of view (fovy) and aspect ratio (aspect).

The documentation for this class was generated from the following file:

```
• MOGL/include/MOGL/Camera.hpp
```

Generated by Doxygen
6.6  

**hum::Clock Class Reference**

Class for measuring intervals of time.

```cpp
#include <Clock.hpp>
```

**Public Member Functions**

- **Clock ()**
  
  Class constructor. Initializes a new instance of **Clock** which starts measuring the time as it is created.

- **~Clock ()**
  
  Class destructor. Destroys the instance of **Clock**.

- **Time getTime () const**
  
  Returns the **Time** elapsed since creation or last reset.

- **Time reset ()**
  
  Restarts the **Time** counter.

6.6.1  Detailed Description

Class for measuring intervals of time.

6.6.2  Member Function Documentation

6.6.2.1  **Time hum::Clock::getTime () const**

Returns the **Time** elapsed since creation or last reset.

Returns

**Time** since instance creation or last **reset**().

6.6.2.2  **Time hum::Clock::reset ()**

Restarts the **Time** counter.

This function also returns the **Time** since instance creation or last **reset**(). This is usefull for getting precise delta times without having time overhead for having to call two methods:

```cpp
hum::Clock clk;
while(game_is_running)
|
| hum::Time deltaTime = clk.reset();
| ... // Game code
|
```

Returns

**Time** since instance creation or last **reset**().

The documentation for this class was generated from the following file:

- **hummingbird/include/hummingbird/Clock.hpp**

Generated by Doxygen
6.7  mogl::Drawable Class Reference

Abstract class to implement a `hum::Behavior` that can be drawn.

```cpp
#include <Drawable.hpp>
```

Inheritance diagram for `mogl::Drawable`:

Collaboration diagram for `mogl::Drawable`:

```
Public Member Functions

- `Drawable()`  
  Constructor.
- `virtual ~Drawable()`  
```
Destructor.

- virtual void init () override
  Behavior initialization function.

- virtual void onActivate () override
  Called when the Actor is activated.

- virtual void onDeactivate () override
  Called when the Actor is deactivated.

- virtual void onDestroy () override
  Called when the Actor is about to be destroyed.

- void enable ()
  Enable the drawable.

- void disable ()
  Disable the drawable.

- bool isEnabled () const
  Query if the Drawable is enabled.

- hum::Transformation & transform ()
  Get a reference to the Drawable's hum::Transformation.

- const hum::Transformation & transform () const
  Get a constant reference to the Drawable's hum::Transformation.

- virtual void setShaderProgram (ShaderProgram * shader_program)
  Set the ShaderProgram to use with the Drawable.

- const ShaderProgram * shaderProgram () const
  Get the ShaderProgram being used by the Drawable.

- ShaderProgram * shaderProgram ()
  Get the ShaderProgram being used by the Drawable.

- virtual void draw ()=0
  Draw the Drawable to the active OpenGL context.

- void setOrigin (const hum::Vector3f & origin)
  Set the origin (center) of the Drawable.

- const hum::Vector3f & getOrigin () const
  Get the origin (center) of the Drawable.

---

Static Public Member Functions

- static const char * behaviorName ()

---

6.7.1 Detailed Description

Abstract class to implement a hum::Behavior that can be drawn.

Drawable is a base class that implements the basic functionalities for drawing geometry into OpenGL. These functionalities include a hum::Transformation that id relative to its hum::Actor one; An origin for defining the center; shaderProgram setter and getters; and the ability to disable and enable it.

Example:
class MyDrawable : public mogl::Drawable
{
public:
  // ...
  // Overwrite setShaderProgram() to bind the vertex attribute

  void setShaderProgram(ShaderProgram* shader_program)
  {
    Drawable::setShaderProgram(shader_program);
    glBindVertexArray(s_VAO);
    glBindBuffer(GL_ARRAY_BUFFER, s_VBO);
    p_position_loc = shaderProgram()->bindVertexAttribute("position", 2, 0, 0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
  }

  // When this is called the shader is already in use and the model, view and
  // projection matrices are set.
  virtual void draw()
  {
    glBindVertexArray(s_VAO);
    shaderProgram()->setUniform4f("color",
                               static_cast<float>(p_color.r)/255.0f,
                               static_cast<float>(p_color.g)/255.0f,
                               static_cast<float>(p_color.b)/255.0f,
                               static_cast<float>(p_color.a)/255.0f);
    glEnableVertexAttribArray(p_position_loc);
    glDrawArrays(GL_TRIANGLES, 0, 6);
  }
};

Shaders for Drawable must have the following uniforms defined:

- mat4 projection
- mat4 view
- mat4 model

Which are the standard matrices to render.

6.7.2 Member Function Documentation

6.7.2.1 void mogl::Drawable::disable()

Disable the drawable.

A Drawable is enabled by default. If disabled it won't be drawn.

6.7.2.2 virtual void mogl::Drawable::draw() [pure virtual]

Draw the Drawable to the active OpenGL context.

This abstract method is to be implemented by derived classes to define how the Drawable must be drawn.

By the point where this method is called, the ShaderProgram of the Drawable is being used and the model, view
and projection uniform matrices are set.

Implemented in mogl::AnimatedSprite, mogl::Sprite, and mogl::Rectangle.
6.7.2.3  void mogl::Drawable::enable ( )

Enable the drawable.

A Drawable is enabled by default. If enabled it will be drawn.

6.7.2.4  virtual void mogl::Drawable::init ( ) [override],[virtual]

Behavior initialization function.

This method is called only once, the first preUpdate of the behavior in the Actor.

This method is useful for initializing values that require accessing other Behaviors in the Actor, as it guarantees that they are accessible from this point on.

It also guarantees that the Game is running and all Plugins’ gameStart() have been called at this point, therefore the Game has been initialized.

Reimplemented from hum::Behavior.

Reimplemented in mogl::Sprite, and mogl::Rectangle.

6.7.2.5  bool mogl::Drawable::isEnabled ( ) const

Query if the Drawable is enabled.

Returns

Whether the Drawable is enabled or not.

6.7.2.6  virtual void mogl::Drawable::onActivate ( ) [override],[virtual]

Called when the Actor is activated.

This method is not called when the Behavior is initialized as it is assumed that a Behavior is only initialized if it’s Actor is active.

Reimplemented from hum::Behavior.

6.7.2.7  virtual void mogl::Drawable::setShaderProgram ( ShaderProgram ∗ shader_program ) [virtual]

Set the ShaderProgram to use with the Drawable.

The Drawable doesn't handle the given pointer and the pointed object must exists while the Drawable is using it.

Reimplemented in mogl::Sprite, and mogl::Rectangle.
6.7.2.8  

hum::Transformation& mogl::Drawable::transform ( )

Get a reference to the Drawable's hum::Transformation.

This hum::Transformation is relative to the Drawable's hum::Actor.

Returns
   Drawable's hum::Transformation..

6.7.2.9  constexpr hum::Transformation& mogl::Drawable::transform ( ) const

Get a constant reference to the Drawable's hum::Transformation.

This hum::Transformation is relative to the Drawable's hum::Actor.

Returns
   Drawable's hum::Transformation..

The documentation for this class was generated from the following file:

   • MOGL/include/MOGL/Drawable.hpp

6.8  

hum::Game Class Reference

The class that runs the Game.

#include <Game.hpp>

Public Member Functions

   • Game (unsigned int fixed_tickrate=60)
      Class constructor.
   • ~Game ()
      Class destructor.
   • void run ()
      Run the Game.
   • const Time & deltaTime () const
      Get the delta Time of the game update.
   • Time fixedUpdateTime () const
      Get the delta Time of the game fixedUpdate.
   • Time fixedUpdateLag () const
      Get the Time since the last fixedUpdate.
   • void setRunning (bool running)
      Set the Game's running status.
   • Actor * makeActor ()
      Create a new Actor instance.
• void destroy (Actor *actor)
  Mark an Actor to be destroyed.
• void destroy (Actor &actor)
  Mark an Actor to be destroyed.
• const std::unordered_set<Actor *> &actors () const
  Get the Actor object pool.
• template<typename P, class... Args>
  P * addPlugin (Args &&...args)
  Add a new Plugin to the Game.
• template<typename P>
  const P * getPlugin () const throw (exception::PluginNotFound)
  Get and existing Plugin from the Game.
• template<typename P>
  P * getPlugin () throw (exception::PluginNotFound)
  Get and existing Plugin from the Game.

6.8.1 Detailed Description

The class that runs the Game.

The Game class handles the game loop and the lifecycle of the game. To start running a Game just call Game::run() and the Game instance will enter the following flow chart. The flow chart shows the lifecycle of a Game and the different steps that happen through it.

![Figure 6.1 Game loop](Generated by Doxygen)
A **Game** instance can be queried for **Time** information at any step (when the **Game** is running). This information can be the `deltaTime()`, which is the **Time** that passed between the previous `update` and the current `update`; the `fixedUpdateTime()` which is the **Time** that passes between `fixedUpdate` and `fixedUpdate` and is calculated from the `fixed_tickrate` constructor parameter; and the `fixedUpdateLag()`, this last value is the **Time** since the last `fixedUpdate`.

The running state of the **Game** can be controlled by using `setRunning()`. If set to `false` the game loop will exit and the **Game** will end. A **Game** instance should not be reused once the **Game** is done running, as the final state is not guaranteed in any way.

The **Game** class also owns the **Actor** object pool and therefore handles the creation and destruction of **Actors**.

**Example code:**

```cpp
hum::Game game;
hum::Actor* new_actor = game.makeActor(); // creation of a new Actor;
// ...
game.destroy(new_actor); // mark the Actor to be destroyed.
```

**Actors** are not destroyed right away, but marked to be destroyed and destroyed after `Plugin::postUpdate()`. All **Actors** are destroyed automatically after `Plugin::gameEnd()`.

A **Game** instance can also contain **Plugins**. **Plugins** can implement functionality for the **Game** such as a rendering pipeline, scene management, etc. They can be added and queried by typename using `addPlugin()` and `getPlugin()` template methods respectively (example below).

```cpp
class MyPlugin : public hum::Plugin {...};
hum::Game game;
MyPlugin* mp = game.addPlugin<MyPlugin>();
// somewhere else in the code (p.e. inside a Behavior)
MyPlugin* mp = game().getPlugin<MyPlugin>();
```

A **Plugin** shouldn't be added after calling `run()`.

### 6.8.2 Constructor & Destructor Documentation

#### 6.8.2.1 `hum::Game::Game ( unsigned int fixed_tickrate = 60 )`

Class constructor.

`fixed_tickrate` is the frequency of the `fixedUpdate`. By default 60Hz.

#### 6.8.2.2 `hum::Game::~Game ( )`

Class destructor.

Takes care of destroying any remaining **Actors** and **Plugins**.

### 6.8.3 Member Function Documentation

#### 6.8.3.1 `const std::unordered_set<Actor*>& hum::Game::actors ( ) const`

Get the **Actor** object pool.

Returns

A set of pointers to all **Actors**.

#### 6.8.3.2 `template<typename P, class... Args> P* hum::Game::addPlugin ( Args &&... args ) [inline]`

Add a new **Plugin** to the **Game**.
Template Parameters

- \( P \) The Plugin's type.
- Args The parameters for \( P \)'s constructor.

Returns

A \( P \) pointer to the new Plugin instance.

6.8.3.3 \texttt{const Time\& hum::Game::deltaTime ( ) const}

Get the delta \texttt{Time} of the game update.

Returns

\texttt{Time} elapsed between previous update and current update.

6.8.3.4 \texttt{Time hum::Game::fixedUpdateLag ( ) const}

Get the \texttt{Time} since the last fixedUpdate.

Returns

\texttt{Time} elapsed since the last fixedUpdate.

6.8.3.5 \texttt{Time hum::Game::fixedUpdateTime ( ) const}

Get the delta \texttt{Time} of the game fixedUpdate.

Returns

\texttt{Time} elapsed between previous fixedUpdate and current fixedUpdate.

6.8.3.6 \texttt{template<typename P > const P* hum::Game::getPlugin ( ) const throw (exception::PluginNotFound)}

Get and existing Plugin from the Game.

Template Parameters

- \( P \) The Plugin's type.

If the Game doesn't contain a Plugin of the given type it will throw a exception::PluginNotFound exception.
Returns

A `P` pointer to the Plugin instance.

### 6.8.3.7 template<typename P> `P*` hum::Game::getPlugin () throw(exception::PluginNotFound) [inline]

Get and existing Plugin from the Game.

**Template Parameters**

- `P` The Plugin's type.

If the Game doesn't contain a Plugin of the given type it will throw a exception::PluginNotFound exception.

Returns

A `P` pointer to the Plugin instance.

### 6.8.3.8 `Actor*` hum::Game::makeActor ()

Create a new Actor instance.

Returns

The new Actor instance's pointer.

### 6.8.3.9 `void` hum::Game::run ()

Run the Game.

Starts running the game loop. See the Game class description for the flow chart of the main loop's steps.

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Game.hpp

---

### 6.9 mogl::InputHandler Class Reference

Class that handles SFML input events and allows for easy querying of related data.

#include <InputHandler.hpp>

Generated by Doxygen
Public Member Functions

- **InputHandler ()**
  
  Class constructor.

- **~InputHandler ()**
  
  Class destructor.

- **void update ()**
  
  Run one update step.

- **void handleEvent (const sf::Event &event)**
  
  Handle a new event.

**Keyboard**

- **bool isKeyPressed (sf::Keyboard::Key key)**
  
  Get whether a given key has been just pressed.

- **bool isKeyDown (sf::Keyboard::Key key)**
  
  Get whether a given key has been kept pressed.

- **bool isKeyReleased (sf::Keyboard::Key key)**
  
  Get whether a given key has been just released.

**Mouse**

- **bool mouseMoved ()**
  
  Get whether the mouse just moved.

- **const hum::Vector2i & getMouseCurrentPosition ()**
  
  Get the current position of the mouse.

- **const hum::Vector2i & getMousePreviousPosition ()**
  
  Get the previous position of the mouse.

- **bool isMouseButtonPressed (sf::Mouse::Button button)**
  
  Get if a given mouse button has been just pressed.

- **bool isMouseButtonDown (sf::Mouse::Button button)**
  
  Get if a given mouse button has been kept pressed.

- **bool isMouseButtonReleased (sf::Mouse::Button button)**
  
  Get if a given mouse button has been just released.

- **bool mouseWheelScrolled ()**
  
  Get if the mouse wheel has just been scrolled.

- **float getMouseWheelScrollDelta ()**
  
  Get the amount that the mouse wheel has been scrolled.

**JoyStick**

- **bool isJoystickConnected (unsigned int joystickId)**
  
  Check whether a JoyStick is connected.

- **bool isJoystickActive (unsigned int joystickId)**
  
  Check whether a JoyStick is active.

- **bool isJoystickDisconnected (unsigned int joystickId)**
  
  Check whether a JoyStick is disconnected.

- **bool isJoystickButtonPressed (unsigned int joystickId, unsigned int button)**
  
  Check whether a button of a JoyStick has been just pressed.

- **bool isJoystickButtonDown (unsigned int joystickId, unsigned int button)**
  
  Check whether a button of a JoyStick has been kept pressed.

- **bool isJoystickButtonReleased (unsigned int joystickId, unsigned int button)**
  
  Check whether a button of a JoyStick has been just released.

- **float getJoystickAxisPosition (unsigned int joystickId, sf::Joystick::Axis axis)**
  
  Get the position of an Axis of a given JoyStick.
6.9.1 Detailed Description

Class that handles SFML input events and allows for easy querying of related data.

6.9.2 Member Function Documentation

6.9.2.1 void mogl::InputHandler::handleEvent ( const sf::Event & event )

Handle a new event.

Every sf::Event polled from the sf::Window must be passed to this method. It takes care of treating it.

6.9.2.2 void mogl::InputHandler::update ( )

Run one update step.

This method must be ran once per logical update cycle as it takes care of updating, for example, when a key that was just pressed passes to being kept pressed.

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/InputHandler.hpp

6.10 hum::Kinematic Class Reference

Add Kinematic properties to the Actor. (Requires KinematicWorld).

#include <Kinematic.hpp>

Inheritance diagram for hum::Kinematic:

```
hum::Behavior

hum::Kinematic
```
Collaboration diagram for hum::Kinematic:

```
hum::Behavior

hum::Kinematic
```

## Public Member Functions

- **Kinematic ()**
  
  Class constructor.

- **virtual ~Kinematic ()**
  
  Class destructor.

- **virtual void init () override**
  
  Behavior initialization function.

- **virtual void onDestroy () override**
  
  Called when the Actor is about to be destroyed.

- **virtual Transformation simulate (const Time &delta_time) const**
  
  Get the transformation of the Actor after applying the kinematic movement during a given time (<delta_time>).

- **Transformation & velocity ()**
  
  Get the velocity of the Kinematic Transformation.

- **const Transformation & velocity () const**
  
  Get the velocity of the Kinematic Transformation.

- **Transformation & acceleration ()**
  
  Get the acceleration of the Kinematic Transformation.

- **const Transformation & acceleration () const**
  
  Get the acceleration of the Kinematic Transformation.

## Static Public Member Functions

- **static const char * behaviorName ()**

### 6.10.1 Detailed Description

Add Kinematic properties to the Actor. (Requires KinematicWorld).

This Behavior allows to give a velocity and an acceleration to an Actor. This way the Actor's Transformation will change automatically overtime following a kinematic movement.

Usage example:

```cpp
hum::Game game;
game.addPlugin<hum::KinematicWorld>();

hum::Actor* actor = game.makeActor();
hum::Kinematic* k = actor->addBehavior<hum::Kinematic>();
k->velocity().position.x = 5;
k->acceleration().rotation.z = 2;
```
6.11 hum::KinematicWorld Class Reference

6.10.2 Member Function Documentation

6.10.2.1 virtual void hum::Kinematic::init ( ) [override],[virtual]

Behavior initialization function.

This method is called only once, the first preUpdate of the behavior in the Actor.

This method is useful for initializing values that require accessing other Behaviors in the Actor, as it guarantees that they are accessible from this point on.

It also guarantees that the Game is running and all Plugins' gameStart() have been called at this point, therefore the Game has been initialized.

Reimplemented from hum::Behavior.

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Kinematic.hpp

6.11 hum::KinematicWorld Class Reference

Plugin that handles the transformation (movement, scale or rotation) of an Actor with a Kinematic behavior.

#include <KinematicWorld.hpp>

Inheritance diagram for hum::KinematicWorld:

```
hum::Plugin

hum::KinematicWorld
```

Collaboration diagram for hum::KinematicWorld:

```
hum::Plugin

hum::KinematicWorld
```
Public Member Functions

- **void add (Kinematic ∗kinematic)**
  
  Add a Kinematic Behavior to handle by the Plugin. (Internal use).

- **void remove (Kinematic ∗kinematic)**
  
  Remove a Kinematic Behavior to handle by the Plugin. (Internal use).

- **void preFixedUpdate () override**
  
  Function called before each fixed update.

Protected Attributes

- **std::set< Kinematic ∗ > p_kinematics**

Additional Inherited Members

6.11.1 Detailed Description

Plugin that handles the transformation (movement, scale or rotation) of an Actor with a Kinematic behavior.

Usage example:

```
hum::Game game;
game.addPlugin<hum::KinematicWorld>();
```

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/KinematicWorld.hpp

6.12 mogl::MultimediaOGL Class Reference

Plugin that handles all the rendering pipeline, input and resource management.

```
#include <MultimediaOGL.hpp>
```

Inheritance diagram for mogl::MultimediaOGL:
Collaboration diagram for mogl::MultimediaOGL:

Public Member Functions

- **MultimediaOGL** (sf::VideoMode mode, const sf::String &title, sf::Uint32 style=sf::Style::Default, const sf::ContextSettings &settings=sf::ContextSettings())
  
  Class constructor.

- virtual ~MultimediaOGL ()
  
  Class destructor.

- virtual void gameStart () override
  
  Function called once on game start.

- virtual void preFixedUpdate () override
  
  Function called before each fixed update.

- virtual void postUpdate () override
  
  Function called after each update.

- virtual void gameEnd () override
  
  Function called once on game end.

- virtual sf::Window * window ()
  
  Get a pointer to the window.

- virtual const sf::Window * window () const
  
  Get a pointer to the window.

- virtual void addDrawable (Drawable *drawable)
  
  Register a Drawable to be drawn (Internal use only).

- virtual void removeDrawable (Drawable *drawable)
  
  Register a Drawable to be drawn (Internal use only).

- virtual void setClearColor (const sf::Color &color)
  
  Set the clear color for OpenGL.

- virtual const sf::Color & getClearColor () const
  
  Get the clear color for OpenGL.

- virtual void setCamera (const Camera &camera)
  
  Set a new Camera to use.

- virtual Camera & getCamera () const
  
  Get the Camera being used.

- virtual Camera & getCamera ()
  
  Get the Camera being used.

- virtual void setDrawSpaceTransform (const SpaceTransformation &space_transform)
  
  Set the space transformation between the game logic space and the draw space.
Resource Managers

- **ShaderProgramManager & shaderPrograms ()**
  Get the ShaderProgramManager.
- **const ShaderProgramManager & shaderPrograms () const**
  Get the ShaderProgramManager.
- **InputHandler & input ()**
  Get the InputHandler.
- **const InputHandler & input () const**
  Get the InputHandler.
- **SpriteAnimationManager & spriteAnimations ()**
  Get the SpriteAnimationManager.
- **const SpriteAnimationManager & spriteAnimations () const**
  Get the SpriteAnimationManager.
- **TextureManager & textures ()**
  Get the TextureManager.
- **const TextureManager & textures () const**
  Get the TextureManager.
- **SoundManager & sounds ()**
  Get the SoundManager.
- **const SoundManager & sounds () const**
  Get the SoundManager.
- **MusicManager & music ()**
  Get the MusicManager.
- **const MusicManager & music () const**
  Get the MusicManager.

Additional Inherited Members

6.12.1 Detailed Description

Plugin that handles all the rendering pipeline, input and resource management.

The MultimediaOGL plugin is the class that groups all the functionalities included in MOGL and makes them accessible in the hum::Game.

Rendering

The plugin handles the creation of the sf::Window and the OpenGL context. The window is made accessible through window().

The plugin also handles the camera, that is accessible by calling getCamera(). To set and get the clear color setClearColor() and getClearColor() can be called.

The rendering of all active and enabled Drawables happens at every postUpdate. This may seem anti-intuitive because the position of the Actors is only updated every fixedUpdate (we’d be drawing the same frame multiple times), but MultimediaOGL does something smart: for all Drawables whose Actors have a hum::Kinematic behavior, the position of the Drawable will be interpolated using the kinematic information and the fixedUpdateLag. Also, AnimatedSprites’s animation is also updated when drawn. This way we can have a smooth as possible view of the game world and a fixedUpdate for the game logic.

Finally, a game space -> draw space transformation can be set by using setDrawSpaceTransform(), for more details see the method’s details.
Resource Management

The `MultimediaOGL` plugin owns an instance of each of the resource managers included with MOGL. This way one can access any of them at any point inside the game.

Example:

```cpp
class Player : public hum::Behavior
{
private:
    hum::Kinematic* k;
    mogl::AnimatedSprite* spr;
    mogl::SoundId sound_id;
    mogl::MultimediaOGL* mogl;
public:
    void init() override
    {
        sound_id = 0;
        mogl = actor().game().getPlugin<mogl::MultimediaOGL>();
        k = actor().getBehavior<hum::Kinematic>();
        spr = actor().getBehavior<mogl::AnimatedSprite>();
        spr->setSpriteAnimation(mogl->spriteAnimations().get("player_idle"));
    }
    void fixedUpdate() override
    {
        ...
        if (k->velocity().x != 0 || k->velocity().y != 0)
        {
            auto anim = mogl->spriteAnimations().get("player_walking");
            if (anim != spr->spriteAnimation())
            {
                sound_id = mogl->sounds()->play("steps", 75, true);
            }
        }
        else
        {
            auto anim = mogl->spriteAnimations().get("player_idle");
            if (anim != spr->spriteAnimation())
            {
                if (mogl->sounds().get(sound_id) != nullptr)
                {
                    mogl->sounds()->get(sound_id)->stop();
                }
            }
        }
    }
}
```

### 6.12.2 Constructor & Destructor Documentation

#### 6.12.2.1 `mogl::MultimediaOGL::MultimediaOGL ( sf::VideoMode mode, const sf::String & title, sf::Uint32 style = sf::Style::Default, const sf::ContextSettings & settings = sf::ContextSettings() )`

Class constructor.

Initializes all resource managers and the OpenGL context. but doesn’t create the window yet.

### 6.12.3 Member Function Documentation

#### 6.12.3.1 `virtual void mogul::MultimediaOGL::setCamera ( const Camera & camera ) [virtual]`

Set a new `Camera` to use.

Usage disencouraged. Better use `getCamera()` and set the desired configuration using its setters.
6.12.3.2 virtual void mogl::MultimediaOGL::setDrawSpaceTransform ( const SpaceTransformation & space_transform )

Set the space transformation between the game logic space and the draw space.

This is useful when, for example you have a game with 2D logic but you want to render it on an irregular terrain (height map). You can use this method for decoupling the 2D logic from the 3D representation.

Another example is to draw a 2D tile based (position is always aligned with the tiles) game as a 2D isometric game.

Example:

```cpp
// Suppose we have a hum::Plugin for managing terrains.
void setTerrainHeight(const hum::Game& game, hum::Transformation& r)
{
    r.position.z = game.getPlugin<Terrains>()->getCurrent().height(r.position.x, r.position.y);
}

//...
game().getPlugin<mogl::MultimediaOGL>()->
    setDrawSpaceTransform(setTerrainHeight);
```

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/MultimediaOGL.hpp

### 6.13 mogl::MusicManager Class Reference

A hum::ResourceManager for sf::Music.

```
#include <MusicManager.hpp>
```

Inheritance diagram for mogl::MusicManager:
Collaboration diagram for mogl::MusicManager:

```
hum::ResourceManager
sf::Music

mogl::MusicManager
```

Protected Member Functions

- `sf::Music * loadResource (const std::string &name) override`
  
  Function to implement for loading the new instance of the resource type.

Additional Inherited Members

6.13.1 Detailed Description

A `hum::ResourceManager` for `sf::Music`.

Example:

```cpp
mogl::MusicManager mm;
mm.load("music1", "path/to/music1.ogg");
sf::Music * music = mm.get("music1");
music->play();
//...
music->stop();
mm.free("music1");
```

6.13.2 Member Function Documentation

6.13.2.1 `sf::Music * mogul::MusicManager::loadResource ( const std::string & data )` [override], [protected], [virtual]

Function to implement for loading the new instance of the resource type.

It receives a Data (by default `std::string`) with information to load the new resource instance.

If there is an error when loading the resource it must return `nullptr`.

Implements `hum::ResourceManager< sf::Music >`.

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/MusicManager.hpp

Generated by Doxygen
6.14  hum::Plugin Class Reference

Class from which inherit to implement and give a Plugin for the Game.

```cpp
#include <Plugin.hpp>
```

Inheritance diagram for hum::Plugin:

```
hum::Plugin

hum::KinematicWorld
mogl::MultimediaOGL
```

Public Member Functions

- virtual void `gameStart` ()
  
  *Function called once on game start.*

- virtual void `preUpdate` ()
  
  *Function called before each update.*

- virtual void `preFixedUpdate` ()
  
  *Function called before each fixed update.*

- virtual void `postFixedUpdate` ()
  
  *Function called after each fixed update.*

- virtual void `postUpdate` ()
  
  *Function called after each update.*

- virtual void `gameEnd` ()
  
  *Function called once on game end.*

Protected Member Functions

- `Game & game` ()
- `const Game & game` () const
6.14.1 Detailed Description

Class from which inherit to implement and give a Plugin for the Game.

A Plugin always lives inside the Game. The Game takes care of the lifecycle of the Plugin.

For creating a custom Plugin you may inherit from this class and override the methods you need to implement the wanted functionality.

For more information on the lifecycle of a Game see the Game class description.

Usage example:

```cpp
class DeltaTimePlugin : public hum::Plugin
{
    void gameStart() override {
        hum::log("Game just started");
    }

    void preUpdate() override {
        hum::log("delta time: ", game().deltaTime());
    }

    void preFixedUpdate() override {
        hum::log("fixed update lag: ", game().fixedUpdateLag());
    }

    void postFixedUpdate() override {
        hum::log("fixed update lag: ", game().fixedUpdateLag());
    }

    void postUpdate() override {
        hum::log("delta time: ", game().deltaTime());
    }

    void gameEnd() override {
        hum::log("Game just finished");
    }
};
```

6.14.2 Member Function Documentation

6.14.2.1 Game& hum::Plugin::game() [protected]

Get the Plugin's Game.

Returns

The Plugin's Game

6.14.2.2 const Game& hum::Plugin::game() const [protected]

Get the Plugin's Game.

Returns

The Plugin's Game

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Plugin.hpp

Generated by Doxygen
6.15  hum::exception::PluginNotFound Class Reference

Exception thrown when getting a Plugin from a Game that does not contain it. (see Game::getPlugin())

#include <Exceptions.hpp>

Inherits std::exception.

Public Member Functions

• virtual const char * what () const noexcept override

6.15.1  Detailed Description

Exception thrown when getting a Plugin from a Game that does not contain it. (see Game::getPlugin())

The documentation for this class was generated from the following file:

• hummingbird/include/hummingbird/Exceptions.hpp

6.16  mogl::Rectangle Class Reference

A rectangle-shaped 1x1 Drawable.

#include <Rectangle.hpp>

Inheritance diagram for mogl::Rectangle:

```
hum::Behavior

mogl::Drawable

mogl::Rectangle
```

Generated by Doxygen
Collaboration diagram for mogl::Rectangle:

```
hum::Behavior

mogl::Drawable

mogl::Rectangle
```

### Public Member Functions

- **Rectangle** (const sf::Color &)
  
  *Class constructor with fill color.*

- void **init** () override
  
  *Behavior initialization function.*

- void **onDestroy** () override
  
  *Called when the Actor is about to be destroyed.*

- void **setShaderProgram** (ShaderProgram *shader_program) override
  
  *Set the ShaderProgram to use with the Drawable.*

- void **setColor** (const sf::Color &color)
  
  *Set fill color for the Rectangle.*

- const sf::Color & **getColor** () const
  
  *Get fill color for the Rectangle.*

- void **draw** () override
  
  *Draw the Rectangle.*

### Static Public Member Functions

- static const char * **behaviorName** ()

### 6.16.1 Detailed Description

A rectangle-shaped 1x1 Drawable.

For different sizes use the scale in either the hum::Actor hum::Transform of the Drawable's hum::Transform.
6.16.2 Member Function Documentation

6.16.2.1 void mogl::Rectangle::init ( ) [override],[virtual]

Behavior initialization function.

This method is called only once, the first preUpdate of the behavior in the Actor.

This method is useful for initializing values that require accessing other Behaviors in the Actor, as it guarantees that they are accessible from this point on.

It also guarantees that the Game is running and all Plugins' gameStart() have been called at this point, therefore the Game has been initialized.

Reimplemented from mogl::Drawable.

6.16.2.2 void mogl::Rectangle::setShaderProgram ( ShaderProgram ∗ shader_program ) [override], [virtual]

Set the ShaderProgram to use with the Drawable.

The Drawable doesn’t handle the given pointer and the pointed object must exists while the Drawable is using it.

Reimplemented from mogl::Drawable.

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/Rectangle.hpp

6.17 hum::ResourceManager＜T, Key, Data＞ Class Template Reference

Class that implements the generic functionality of a resource manager.

#include ＜ResourceManager.hpp＞

Public Member Functions

- virtual bool load (const Key &key, const Data &data)
  Loads a resource from ＜data＞, if not already loaded.
- virtual T ∗ get (const Key &key)
  Get a pointer to the resource identified by ＜key＞. If not loaded returns nullptr.
- virtual const T ∗ get (const Key &key) const
  Get a pointer to the resource identified by ＜key＞. If not loaded returns nullptr.
- virtual void free (const Key &key)
  Free the resource identified by ＜key＞. If not loaded it does nothing.

Protected Member Functions

- virtual T ∗ loadResource (const Data &data)=0
  Function to implement for loading the new instance of the resource type.
6.17 hum::ResourceManager< T, Key, Data > Class Template Reference

Protected Attributes

- std::unordered_map<Key, T*> p_resources

6.17.1 Detailed Description

template<typename T, typename Key = std::string, typename Data = std::string>
class hum::ResourceManager< T, Key, Data >

Class that implements the generic functionality of a resource manager.

This template class has three type parameters, two of which are optional. The first is the type of the data to manage. The second one is the type of the key to identify the managed data (std::string by default). The third is the data needed to load the resource (std::string by default).

Usage example (SFML Texture manager):

class TextureManager : public ResourceManager<sf::Texture>
{
protected:
  sf::Texture* loadResource(const std::string& name) override
  {
    sf::Texture* resource = new sf::Texture();
    if (not resource->loadFromFile(name))
      return nullptr;
    return resource;
  }
};

//...
TextureManager tm;
if (!tm.load("cat", "cat.jpg")) {
  hum::log_e("Error loading cat.jpg");
}
if (!tm.load("dog", "dog.jpg")) {
  hum::log_e("Error loading dog.jpg");
}
sf::Texture* cat = tm.get("cat"); // get the texture
tm.free("cat"); // unload the cat texture manually
// when destroyed the resource manager will free all loaded resources.

6.17.2 Member Function Documentation

6.17.2.1 template<typename T, typename Key = std::string, typename Data = std::string> virtual bool
hum::ResourceManager< T, Key, Data >::load ( const Key & key, const Data & data ) [inline],
[virtual]

Loads a resource from <data>, if not already loaded.

Returns true if resource is loaded and can be accessed and returns false otherwise. Sets <key> as the name of the resource.
6.17.2.2 template<typename T, typename Key = std::string, typename Data = std::string> virtual T*
  hum::ResourceManager<T, Key, Data>::loadResource(const Data& data) [protected], [pure
  virtual]

Function to implement for loading the new instance of the resource type.
It receives a Data (by default std::string) with information to load the new resource instance.
If there is an error when loading the resource it must return nullptr.
Implemented in mogl::SoundManager, mogl::MusicManager, and mogl::TextureManager.
The documentation for this class was generated from the following file:

  • hummingbird/include/hummingbird/ResourceManager.hpp

6.18 mogl::Shader Class Reference

Class for loading a Vertex Shader or a Fragment Shader.

#include <Shader.hpp>

Public Types

  • enum Type { VERTEX_SHADER, FRAGMENT_SHADER }
  Type of a Shader.

Public Member Functions

  • Shader()
  Class constructor.
  • ~Shader()
  Class destructor.
  • void loadFromSource(const Type type, const std::string &source)
  Load a shader of Shader::Type <type> from a std::string containing the source.
  • bool loadFromFile(const Type type, const std::string &filename)
  Load a shader of Shader::Type <type> from the file <filename>
  • GLuint getId() const
  Get the native handler of the shader.
  • bool isCompiled() const
  Get whether the Shader code compiled after loading it.
  • const std::string & log() const
  Get the error log of the shader code compilation.

6.18.1 Detailed Description

Class for loading a Vertex Shader or a Fragment Shader.

To be used with ShaderProgram.
6.18.2 Member Function Documentation

6.18.2.1 bool mogl::Shader::isCompiled ( ) const

Get whether the Shader code compiled after loading it.

If this method returns false check log() to get the error.

Returns if the shader code compiled.

6.18.2.2 bool mogl::Shader::loadFromFile ( const Type type, const std::string & filename )

Load a shader of Shader::Type <type> from the file <filename>

This method must be called with an active OpenGL context.

Returns Whether there was and error reading the file.

6.18.2.3 void mogl::Shader::loadFromSource ( const Type type, const std::string & source )

Load a shader of Shader::Type <type> from a std::string containing the source.

This method must be called with an active OpenGL context.

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/Shader.hpp

6.19 mogl::ShaderProgram Class Reference

Class to load, handle and work with shader programs.

#include <ShaderProgram.hpp>
Public Member Functions

- **ShaderProgram ()**
  
  Class constructor.

- **~ShaderProgram ()**
  
  Class destructor.

- **ShaderProgram * addShader (const Shader &shader)**
  
  Add a Shader to the ShaderProgram.

- **ShaderProgram * bindFragmentOutput (const std::string &output_name)**
  
  Bind the output of the fragment shader.

- **GLint bindVertexAttribute (const std::string &attrib_name, GLint size, GLsizei stride, GLvoid *first_pointer)**
  
  Bind a vertex attribute to the ShaderProgram.

- **ShaderProgram * link ()**
  
  Link the various Shaders added into the ShaderProgram.

- **ShaderProgram * use ()**
  
  Set the ShaderProgram as the one currently being used.

- **ShaderProgram * setUniform2f (const std::string &uniform_name, float v0, float v1)**
  
  Pass a vec2 uniform with name <uniform_name> to the associated shaders.

- **ShaderProgram * setUniform3f (const std::string &uniform_name, float v0, float v1, float v2)**
  
  Pass a vec3 uniform with name <uniform_name> to the associated shaders.

- **ShaderProgram * setUniform4f (const std::string &uniform_name, float v0, float v1, float v2, float v3)**
  
  Pass a vec4 uniform with name <uniform_name> to the associated shaders.

- **ShaderProgram * setUniformMatrix4f (const std::string &uniform_name, const glm::mat4 &mat)**
  
  Pass a mat4 uniform with name <uniform_name> to the associated shaders.

- **bool isLinked ()**
  
  Get whether the ShaderProgram was able to link all the associated Shaders.

- **const std::string & log () const**
  
  Get the error log of the shader code compilation.

### 6.19.1 Detailed Description

Class to load, handle and work with shader programs.

A ShaderProgram requires two Shaders to be added (addShader()) to it before linking, one of Shader::Type VERTEX_SHADER and one of Shader::Type FRAGMENT_SHADER.

Example:

```cpp
// Create shaders
mogl::Shader vs, fs;

// Load vertex shader from file
vs.loadFromFile(Shader::Type::VERTEX_SHADER, "shader.vert");
if(!vs.isCompiled())
{
    hum::log_e("Vertex shader failed to compile: ", vs.log());
    return 1;
}

// Load fragment shader from file
fs.loadFromFile(Shader::Type::FRAGMENT_SHADER, "shader.frag");
if(!fs.isCompiled())
{
    hum::log_e("Fragment shader failed to compile: ", fs.log());
    return 1;
}

// Create shader program
mogl::ShaderProgram prog;
```
// Add shaders and link
prog.addShader(vs)
->addShader(fs)
->link();

// Check for errors
if (!prog.isLinked())
{
    hum::log_e("Shader program failed to link: ", prog.log());
    return 1;
}

//...  
prog.use();
prog.setUniform4f("color", 0.f, 1.f, 1.f, 1.f);
//...

Shaders for **Drawable**s must have the following uniforms defined:

- mat4 projection
- mat4 view
- mat4 model

Which are the standard matrices to render.

### 6.19.2 Member Function Documentation

#### 6.19.2.1 ShaderProgram\* mogl::ShaderProgram::addShader ( const \_Shader & \_shader )

Add a **Shader** to the **ShaderProgram**.

Returns

A pointer to itself.

#### 6.19.2.2 ShaderProgram\* mogl::ShaderProgram::bindFragmentOutput ( const std::string & \_output_name )

Bind the output of the fragment shader.

Returns

A pointer to itself.

#### 6.19.2.3 GLint mogl::ShaderProgram::bindVertexAttribute ( const std::string & \_attrib_name, GLsizei \_size, GLsizei \_stride, GLvoid * \_first_pointer )

Bind a vertex attribute to the **ShaderProgram**.

Returns

A pointer to itself.
6.19.2.4 bool mogl::ShaderProgram::isLinked ( )

Get whether the ShaderProgram was able to link all the associated Shaders.

If this method returns false check log() to get the error.

Returns
  if the ShaderProgram linked.

6.19.2.5 ShaderProgram* mogl::ShaderProgram::link ( )

Link the various Shaders added into the ShaderProgram.

Returns
  A pointer to itself.

6.19.2.6 ShaderProgram* mogl::ShaderProgram::setUniform2f ( const std::string & uniform_name, float v0, float v1 )

Pass a vec2 uniform with name <uniform_name> to the associated shaders.

Returns
  A pointer to itself.

6.19.2.7 ShaderProgram* mogl::ShaderProgram::setUniform3f ( const std::string & uniform_name, float v0, float v1, float v2 )

Pass a vec3 uniform with name <uniform_name> to the associated shaders.

Returns
  A pointer to itself.

6.19.2.8 ShaderProgram* mogl::ShaderProgram::setUniform4f ( const std::string & uniform_name, float v0, float v1, float v2, float v3 )

Pass a vec4 uniform with name <uniform_name> to the associated shaders.

Returns
  A pointer to itself.
6.19.2.9 ShaderProgram∗ mogl::ShaderProgram::setUniformMatrix4f ( const std::string & uniform_name, const glm::mat4 & mat )

Pass a mat4 uniform wth name <uniform_name> to the associated shaders.

Returns
A pointer to itself.

6.19.2.10 ShaderProgram∗ mogl::ShaderProgram::use ( )

Set the ShaderProgram as the one currently being used.

Returns
A pointer to itself.

The documentation for this class was generated from the following file:

• MOGL/include/MOGL/ShaderProgram.hpp

6.20 mogl::ShaderProgramManager Class Reference

A hum::ResourceManager for mogl::ShaderProgram.

#include <ShaderProgramManager.hpp>

Inheritance diagram for mogl::ShaderProgramManager:
Collaboration diagram for `mogl::ShaderProgramManager`:

```
hum::ResourceManager
  ShaderProgram, std::string, ShaderProgramDef

mogl::ShaderProgramManager
```

### Additional Inherited Members

#### 6.20.1 Detailed Description

A `hum::ResourceManager` for `mogl::ShaderProgram`.

This Resource manager is different because it uses `ShaderProgramDefs` to load the resource (`ShaderProgram`) instead of the usual `std::string`.

Example:

```cpp
// Create shaders
mogl::Shader vs, fs;

// Load vertex shader from file
vs.loadFromFile(Shader::Type::VERTEX_SHADER, "shader.vert");
if(!vs.isCompiled())
  hum::log_e("Vertex shader failed to compile: ", vs.log());
  return 1;

// Load fragment shader from file
fs.loadFromFile(Shader::Type::FRAGMENT_SHADER, "shader.frag");
if(!fs.isCompiled())
  hum::log_e("Fragment shader failed to compile: ", fs.log());
  return 1;

mogl::ShaderProgramDef def{vs, fs, "out_color"};

mogl::ShaderProgramManager spm;
spm.load("plain_shader", def);

mogl::ShaderProgram* sp = spm.get("plain_shader");
sp->use() -> setUniform4f("color", 0, 1, 1, 1);
// ...
spm.free("plain_shader");
```

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/ShaderProgramManager.hpp
6.21 mogl::SoundManager Class Reference

A hum::ResourceManager for sf::SoundBuffers.

#include <SoundManager.hpp>

Inheritance diagram for mogl::SoundManager:

Collaboration diagram for mogl::SoundManager:

Public Member Functions

- SoundManager ()
  Class constructor.
- ~SoundManager ()
  Class destructor.
- SoundPair play (const std::string &name, int volume, bool loop=false, bool relative_to_listener=false, const sf::Vector3f &position=sf::Vector3f())
  Play a sound.
- sf::Sound * get (SoundId sound_id)
  Get a sound resource by id.
- const sf::Sound * get (SoundId sound_id) const
  Get a sound resource by id.
- void clearSounds ()
  Clear sound resources that are done playing.
Protected Member Functions

- `sf::SoundBuffer * loadResource (const std::string &name)` override
  
  Function to implement for loading the new instance of the resource type.

Additional Inherited Members

6.21.1 Detailed Description

A `hum::ResourceManager` for `sf::SoundBuffers`.

This Resource manager is different because it overwrites the `get()` method and has other extra methods.

The `SoundManager` not just loads `sf::SoundBuffers` but also allows to play them through the method `play()`. This method returns a `std::pair` containing the id of the sound resource used (the one that is playing the required `sf::SoundBuffer`) and a pointer to the `sf::Sound` managing the playback of the `sf::SoundBuffer`. Note that a sound resource is not the same as a `sf::SoundBuffer`.

The `get()` methods are also different. In this manager they are accessed by sound ids (which are returned by `play()`) and they return a pointer of the given `sf::Sound`, or `nullptr` if the sound has been freed.

Example:

```cpp
mogl::SoundManager sm;
sm.load("roar", "path/to/roar.ogg");

//...

// fixedUpdate()
if (event)
|
| roar_id = sm.play("roar", 50).first; // start playing the "roar" sound

// onDestroy()
if (sm.get(roar_id) != nullptr) // if the sound is still playing
|
| sm.get(roar_id)->stop(); // stop it

//...
sm.free("roar");
```

As shown in the example, when a `sf::SoundBuffer` playback is done (Stopped), then the sound resource is cleared and therefore the sound id invalidated (getting its related sound returns `nullptr`).

Sound ids start at 1 and always grow, that means that a sound id of 0 will always return `nullptr`.

6.21.2 Member Function Documentation

6.21.2.1 `void mogl::SoundManager::clearSounds ( )`

Clear sound resources that are done playing.

This is called every `preFixedUpdate` and cleans all sounds that are done playing the `sf::SoundBuffer` they were required to play.
6.21.2.2 sf::Sound * mogl::SoundManager::get ( SoundId sound_id )

Get a sound resource by id.

If the sound resource does not exists, as in it has been freed because the playback of the SoundBuffer is done, it returns nullptr.

6.21.2.3 const sf::Sound * mogl::SoundManager::get ( SoundId sound_id ) const

Get a sound resource by id.

If the sound resource does not exists, as in it has been freed because the playback of the SoundBuffer is done, it returns nullptr.

6.21.2.4 sf::SoundBuffer * mogl::SoundManager::loadResource ( const std::string & data ) [override], [protected], [virtual]

Function to implement for loading the new instance of the resource type.

It receives a Data (by default std::string) with information to load the new resource instance.

If there is an error when loading the resource it must return nullptr.

Implements hum::ResourceManager< sf::SoundBuffer >.

6.21.2.5 SoundPair mogl::SoundManager::play ( const std::string & name, int volume, bool loop = false, bool relative_to_listener = false, const sf::Vector3f & position = sf::Vector3f() )

Play a sound.

Starts playing the sf::SoundBuffer identified by <name> with the given <volume>. It allows to set the sound to loop, whether the sound is relative to the listener and the initial position of the sound.

To set the position of the listener you may use: sf::Listener::setPosition(x, y, z);

For sounds with varying positions, you may do the following:

```cpp
// fixedUpdate
auto sm = actor().game().getPlugin<mogl::MultimediaOGL>()->sounds();
sf::Sound* sound = sm->get(sound_id);
if(sound != nullptr) {
  sound->setPosition(actor().transform().x, actor().transform().y, actor().transform().z);
}
```

Note that the validity of a sf::Sound pointer obtained by play() or get() is not guaranteed between update cycles of the game. (They might get invalidated by clearSounds()).

Returns

a std::pair containig a SoundId (unsigned int) and a pointer to the sf::Sound that is playing the sf::SoundBuffer.

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/SoundManager.hpp

Generated by Doxygen
6.22 mogl::Sprite Class Reference

A textured rectangle-shaped 1x1 Drawable.

#include <Sprite.hpp>

Inheritance diagram for mogl::Sprite:

Collaboration diagram for mogl::Sprite:
Public Member Functions

- **Sprite** (const sf::Texture *texture, const sf::IntRect &rect=sf::IntRect())
  Class constructor with the texture to use and the area of the texture to use.
- void **init** () override
  Behavior initialization function.
- void **onDestroy** () override
  Called when the Actor is about to be destroyed.
- void **setShaderProgram** (ShaderProgram *shader_program) override
  Set the ShaderProgram to use with the Drawable.
- void **setTexture** (const sf::Texture *texture)
  Set the texture to use for the sprite.
- const sf::Texture * **getTexture** () const
  Get the texture to use for the sprite.
- void **setTextureRect** (const sf::IntRect &rect)
  Set the texture area to use for the sprite.
- const sf::IntRect & **getTextureRect** () const
  Get the texture area to use for the sprite.
- void **setColor** (const sf::Color &color)
  Set the color to use for the sprite.
- const sf::Color & **getColor** () const
  Get the color to use for the sprite.
- void **draw** () override
  Draw the Drawable to the active OpenGL context.

Static Public Member Functions

- static const char * **behaviorName** ()

Protected Attributes

- GLuint **p_position_loc**
- GLuint **p_texture_coord_loc**
- sf::Color **p_color**
- const sf::Texture * **p_texture**
- sf::IntRect **p_rect**

Static Protected Attributes

- static GLuint **s_VAO**
- static GLuint **s_VBO**

6.22.1 Detailed Description

A textured rectangle-shaped 1x1 Drawable.

For different sizes use the scale in either the hum::Actor hum::Transform of the Drawable's hum::Transform.
6.22.2 Constructor & Destructor Documentation

6.22.2.1 `mogl::Sprite::Sprite (const sf::Texture * texture, const sf::IntRect & rect = sf::IntRect())`

Class constructor with the texture to use and the area of the texture to use.

If the area is empty or not given the hole texture will be used.

The `Sprite` doesn't handle the given texture pointer and the pointed texture must exists while the `Sprite` is using it.

6.22.3 Member Function Documentation

6.22.3.1 `void mogl::Sprite::draw()` [override], [virtual]

Draw the `Drawable` to the active OpenGL context.

This abstract method is to be implemented by derived classes to define how the `Drawable` must be drawn.

By the point where this method is called, the `ShaderProgram` of the `Drawable` is being used and the `model`, `view` and `projection` uniform matrices are set.

Implements `mogl::Drawable`.

6.22.3.2 `void mogl::Sprite::init()` [override], [virtual]

Behavior initialization function.

This method is called only once, the first preUpdate of the behavior in the Actor.

This method is useful for initializing values that require accessing other Behaviors in the Actor, as it guarantees that they are accessible from this point on.

It also guarantees that the Game is running and all Plugins’ gameStart() have been called at this point, therefore the Game has been initialized.

Reimplemented from `mogl::Drawable`.

6.22.3.3 `void mogl::Sprite::setShaderProgram (ShaderProgram * shader_program)` [override], [virtual]

Set the `ShaderProgram` to use with the `Drawable`.

The `Drawable` doesn't handle the given pointer and the pointed object must exists while the `Drawable` is using it.

Reimplemented from `mogl::Drawable`.
6.23 mogl::SpriteAnimationManager Class Reference

A hum::ResourceManager for mogl::SpriteAnimations.

#include <SpriteAnimationManager.hpp>

Inheritance diagram for mogl::SpriteAnimationManager:

Collaboration diagram for mogl::SpriteAnimationManager:
Additional Inherited Members

6.23.1 Detailed Description

A `hum::ResourceManager` for `mogl::SpriteAnimations`.

This Resource manager is different because it uses `SpriteAnimations` as the data to "load" and stores a copy of the given `SpriteAnimation`.

Example:

```cpp
mogl::SpriteAnimation jump{
game().getPlugin<mogl::MultimediaOGL>()->textures().get("player"),
0, 0, 0, 0, 48, 48,
{5, 6, 7, 8},
std::vector(4, hum::Time::seconds(0.3f))
};
mogl::SpriteAnimationManager sam;
sam.load("player_jump", jump);
//...
actor.addBehavior<mogl::AnimatedSprite>(sam.get("player_jump"));
//...
sam.free("player_jump");
```

The documentation for this class was generated from the following file:

- MOGL/include/MOGL/SpriteAnimationManager.hpp

6.24 `mogl::TextureManager` Class Reference

A `hum::ResourceManager` for `sf::Texture`.

#include <TextureManager.hpp>

Inheritance diagram for `mogl::TextureManager`:

```
hum::ResourceManager
  sf::Texture
  mogl::TextureManager
```
Collaboration diagram for `mogl::TextureManager`:

```
hum::ResourceManager
< sf::Texture >

mogl::TextureManager
```

**Protected Member Functions**

- `sf::Texture * loadResource (const std::string &name) override`  
  Function to implement for loading the new instance of the resource type.

**Additional Inherited Members**

**6.24.1 Detailed Description**

A `hum::ResourceManager` for `sf::Texture`.

Example:

```cpp
mogl::TextureManager tm;
tm.load("cat", "path/to/cat.jpg");
sf::Texture* cat = tm.get("cat");
//...
actor.addBehavior<mogl::Sprite>{cat};
//...
tm.free("cat");
```

**6.24.2 Member Function Documentation**

**6.24.2.1 sf::Texture * mogl::TextureManager::loadResource ( const std::string & data )**  
[override],[protected],[virtual]

Function to implement for loading the new instance of the resource type.

It receives a Data (by default `std::string`) with information to load the new resource instance.

If there is an error when loading the resource it must return `nullptr`.

Implements `hum::ResourceManager< sf::Texture >`.

The documentation for this class was generated from the following file:

```
• MOGL/include/MOGL/TextureManager.hpp
```

Generated by Doxygen
Class for representing intervals of time. This class has nanoseconds precision.

#include <Time.hpp>

Public Member Functions

• Time ()
  Default constructor.
• Time (const Time &other)
  Copy constructor.
• ~Time ()
  Class destructor.
• float asSeconds () const
  Get Time as seconds.
• float asMilliseconds () const
  Get Time as milliseconds.
• float asMicroseconds () const
  Get Time as microseconds.
• long asNanoseconds () const
  Get Time as nanoseconds.
• Time & operator+= (const Time &other)
• Time & operator-= (const Time &other)
• Time & operator*= (float other)
• Time & operator*= (long other)
• Time & operator/= (float other)
• Time & operator/= (long other)
• Time & operator%=(const Time &other)

Static Public Member Functions

• static Time seconds (float seconds)
  Construct Time from seconds.
• static Time milliseconds (float milliseconds)
  Construct Time from milliseconds.
• static Time microseconds (float microseconds)
  Construct Time from microseconds.
• static Time nanoseconds (long nanoseconds)
  Construct Time from nanoseconds.

6.25.1 Detailed Description

Class for representing intervals of time. This class has nanoseconds precision.
6.25.2 Member Function Documentation

6.25.2.1 float hum::Time::asMicroseconds ( ) const

Get Time as microseconds.

Returns

Number of microseconds represented.

6.25.2.2 float hum::Time::asMilliseconds ( ) const

Get Time as milliseconds.

Returns

Number of milliseconds represented.

6.25.2.3 long hum::Time::asNanoseconds ( ) const

Get Time as nanoseconds.

Returns

Number of nanoseconds represented.

6.25.2.4 float hum::Time::asSeconds ( ) const

Get Time as seconds.

Returns

Number of seconds represented.

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Time.hpp

6.26 hum::Transformation Class Reference

Class representing 3D spatial transformation of an object.

#include <Transformation.hpp>

Generated by Doxygen
Public Member Functions

- **Transformation ()**
  
  *Class constructor.*

- **Transformation transform (const Transformation &t) const**
  
  *Transform this Transformation by applying t.*

Public Attributes

- **Vector3f position**
  
  *position vector.*

- **Vector3f rotation**
  
  *rotation degrees around each axis.*

- **Vector3f scale**
  
  *scale for each axis.*

6.26.1 Detailed Description

Class representing 3D spacial transformation of an object.

`hum::Transformation` is a simple class that defines a spacial tranformation of an object. That is it defines a 3D translation, rotation and scale, using `hum::Vector3`.

The `hum::Transformation` class has a small and simple interface, its position, rotation and scale members can be accessed directly (there are no accessors like `setPosition()`, `getPosition()`) and it contains no mathematical function other than the method `transform()`, which accumulates transformations.

Usage example:

```cpp
hum::Transformation t, t2;
t.position.x = 10.f;
t.rotation.z = 90.f;
t.scale.x = 0.5f;

t2.position.x = 5.f;
t2.rotation.z = -25.f;
t2.scale.x = 0.2f;

t = t.transform(t2);

hum::log(t); // hum::Transformation ( position=hum::Vector3( 15, 0, 0 ); rotation=hum::Vector3( 0, 0, 65 ); scale=hum::Vector3( 0.1, 1, 1 ) )
```

6.26.2 Constructor & Destructor Documentation

6.26.2.1 **hum::Transformation::Transformation ( )**

*Class constructor.*

Initializes its members as follows:

```cpp
position = hum::Vector3f(0.f);
rotation = hum::Vector3f(0.f);
scale = hum::Vector3f(1.f);
```

Generated by Doxygen
6.26.3 Member Function Documentation

6.26.3.1 Transformation hum::Transformation::transform ( const Transformation & t ) const

Transform this Transformation by applying t.

This does the equivalent of the following:

```cpp
hum::Transformation t1, t2, result;
result.position = t1.position + t2.position;
result.rotation = t1.rotation + t2.rotation;
result.scale.x = t1.scale.x * t2.scale.x;
result.scale.y = t1.scale.y * t2.scale.y;
result.scale.z = t1.scale.z * t2.scale.z;
```

Returns

The accumulated transformation of this and t.

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Transformation.hpp

6.27 hum::Vector2< T > Class Template Reference

Class representing a 2D vector.

#include <Vector2.hpp>

Public Member Functions

- Vector2 (T x)
  Class constructor. All components are initialized with the given value.
- Vector2 (T x, T y)
  Class constructor by specifying each component's value.
- template<typename U>
  Vector2 (const Vector2< U > &v)
  Cast constructor from other Vector2 of different type.
- T & operator[] (unsigned int position)
  Indexed accessor for the components of the vector.
- const T & operator[] (unsigned int position) const
  Indexed accessor for the components of the vector.

Public Attributes

- T x
- T y

Generated by Doxygen
Related Functions

(Note that these are not member functions.)

• typedef Vector2<float> Vector2f
  A 2D vector of floats.
• typedef Vector2<int> Vector2i
  A 2D vector of ints.
• template<typename T>
  hum::Vector2<T> operator- (const hum::Vector2<T> &right)
  Overload of unary negative operator.
• template<typename T>
  hum::Vector2<T> & operator+= (hum::Vector2<T> &left, const hum::Vector2<T> &right)
  Overload of binary plus-equal operator.
• template<typename T>
  hum::Vector2<T> & operator-= (hum::Vector2<T> &left, const hum::Vector2<T> &right)
  Overload of binary minus-equal operator.
• template<typename T>
  hum::Vector2<T> operator+ (const hum::Vector2<T> &left, const hum::Vector2<T> &right)
  Overload of binary plus operator.
• template<typename T>
  hum::Vector2<T> operator- (const hum::Vector2<T> &left, const hum::Vector2<T> &right)
  Overload of binary minus operator.
• template<typename T>
  hum::Vector2<T> & operator*= (hum::Vector2<T> &left, T right)
  Overload of binary multiply-equal operator.
• template<typename T>
  hum::Vector2<T> & operator/= (hum::Vector2<T> &left, T right)
  Overload of binary divide-equal operator.
• template<typename T>
  hum::Vector2<T> operator* (const hum::Vector2<T> &right)
  Overload of binary multiplication operator.
• template<typename T>
  hum::Vector2<T> operator/ (const hum::Vector2<T> &right)
  Overload of binary multiplication operator.
• template<typename T>
  bool operator==(const hum::Vector2<T> &left, const hum::Vector2<T> &right)
  Overload of binary equal operator.
• template<typename T>
  bool operator!=(const hum::Vector2<T> &left, const hum::Vector2<T> &right)
  Overload of binary different operator.
6.27 hum::Vector2< T > Class Template Reference

6.27.1 Detailed Description

template<typename T>
class hum::Vector2< T >

Class representing a 2D vector.

hum::Vector2 is a simple class that defines a mathematical vector with two coordinates (x and y). It can be used to represent anything that has two dimensions: a size, a point, a velocity, etc.

The template parameter T is the type of the coordinates. It can be any type that supports arithmetic operations (+, -, /, *) and comparisons (==, !=), for example int or float.

You generally don’t have to care about the templated form (hum::Vector2<T>), the most common specializations have special typedefs:

- hum::Vector2<float> is hum::Vector2f
- hum::Vector2<int> is hum::Vector2i

The hum::Vector2 class has a small and simple interface, its x and y members can be accessed directly (there are no accessors like setX(), getX()) and it contains no mathematical function like dot product, cross product, length, etc.

Usage example:

hum::Vector2f v1(16.5f, 24.f);
v1.x = 18.2f;
float y = v1.y;
hum::Vector2f v2 = v1 * 5.f;
hum::Vector2f v3;
v3 = v1 + v2;
bool different = (v2 != v3);

Note: for 3-dimensional vectors, see hum::Vector3

6.27.2 Constructor & Destructor Documentation

6.27.2.1 template<typename T> template<typename U> hum::Vector2<T>::Vector2( const Vector2<U>& v ) [inline]

Cast constructor from other Vector2 of different type.

It will work always that there is a defined cast from U to T.

6.27.3 Member Function Documentation

6.27.3.1 template<typename T> T& hum::Vector2<T>::operator[]( unsigned int position ) [inline]

Indexed accessor for the components of the vector.

Gets the corresponding component as if the vector was an array like [x, y].

Returns

The corresponding component.
Indexed accessor for the components of the vector.

Gets the corresponding component as if the vector was an array like [x, y].

Returns

The corresponding component.

### 6.27.4 Friends And Related Function Documentation

**6.27.4.1 template <typename T> bool operator!= ( const hum::Vector2<T>& left, const hum::Vector2<T>& right )**

Overload of binary different operator.

**Parameters**

<table>
<thead>
<tr>
<th>left</th>
<th>Left vector.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right vector.</td>
</tr>
</tbody>
</table>

Returns

Whether there is a difference between left and right's components.

**6.27.4.2 template <typename T> hum::Vector2<T> operator* ( const hum::Vector2<T>& left, T right )**

Overload of binary multiplication operator.

**Parameters**

<table>
<thead>
<tr>
<th>left</th>
<th>Left factor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right factor.</td>
</tr>
</tbody>
</table>

Returns

The result of multiplying both factors, component by component.

**6.27.4.3 template <typename T> hum::Vector2<T> operator* ( T left, const hum::Vector2<T>& right )**

Overload of binary multiplication operator.
Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left factor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right factor.</td>
</tr>
</tbody>
</table>

Returns

The result of multiplying both factors, component by component.

6.27.4.4 template<typename T> hum::Vector2<T> & operator*= ( hum::Vector2<T> & left, T right ) [related]

Overload of binary multiply-equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left factor and receiver of the result.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right factor.</td>
</tr>
</tbody>
</table>

Returns

A reference to left after having multiplied both factors, component by component, and assigned the result to left.

6.27.4.5 template<typename T> hum::Vector2<T> operator+ ( const hum::Vector2<T> & left, const hum::Vector2<T> & right ) [related]

Overload of binary plus operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left addend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right addend</td>
</tr>
</tbody>
</table>

Returns

The result of adding both addends, component by component.

6.27.4.6 template<typename T> hum::Vector2<T> & operator+= ( hum::Vector2<T> & left, const hum::Vector2<T> & right ) [related]

Overload of binary plus-equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Vector that will be added the value of right.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Value to be added to left.</td>
</tr>
</tbody>
</table>
Returns
A reference to left after having added right to it.

6.27.4.7  template<typename T > hum::Vector2<T> operator- ( const hum::Vector2<T> &right )  [related]

Overload of unary negative operator.
Parameters

| right   | Vector to negate. |

Returns
A copy of the input vector with all its components negated.

6.27.4.8  template<typename T > hum::Vector2<T> operator- ( const hum::Vector2<T> &left, const hum::Vector2<T> &right )  [related]

Overload of binary minus operator.
Parameters

| left     | Left addend.     |
| right    | Right addend.    |

Returns
The result of substracting both addends, component by component.

6.27.4.9  template<typename T > hum::Vector2<T> &operator-= ( hum::Vector2<T> &left, const hum::Vector2<T> &right )  [related]

Overload of binary minus-equal operator.
Parameters

| left     | Vector that will be substracted the value of right. |
| right    | Value to be substracted to left.                   |

Returns
A reference to left after having substracted right to it.
template<typename T> hum::Vector2<T> operator/ ( const hum::Vector2<T> & left, T right )

Overload of binary multiplication operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Dividend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Divisor.</td>
</tr>
</tbody>
</table>

Returns

The quotient of the division, component by component.

template<typename T> hum::Vector2<T> & operator//= ( hum::Vector2<T> & left, T right )

Overload of binary divide-equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Dividend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Divisor.</td>
</tr>
</tbody>
</table>

Returns

A reference to left after having been divided, component by component, by right.

template<typename T> bool operator==( const hum::Vector2<T> & left, const hum::Vector2<T> & right )

Overload of binary equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left vector.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right vector.</td>
</tr>
</tbody>
</table>

Returns

Whether left and right are equal, component by component.

The documentation for this class was generated from the following file:

• hummingbird/include/hummingbird/Vector2.hpp

Generated by Doxygen
6.28  hum::Vector3< T >  Class Template Reference

Class representing a 3D vector.

#include <Vector3.hpp>

Public Member Functions

- **Vector3 (T x)**
  
  Class constructor. All components are initialized with the given value.

- **Vector3 (T x, T y, T z)**
  
  Class constructor by specifying each component's value.

- **Vector3 (const Vector2< T >&xy, T z)**
  
  Class constructor from a Vector2 and a third component.

- **template<typename U> Vector3< U > Vector3 (const Vector3< U >&v)**
  
  Cast constructor from other Vector3 of different type.

- **T & operator[] (unsigned int position)**

  Indexed accessor for the components of the vector.

- **const T & operator[] (unsigned int position) const**

  Indexed accessor for the components of the vector.

Public Attributes

- **T x**
- **T y**
- **T z**

Related Functions

(Note that these are not member functions.)

- **typedef Vector3< float > Vector3f**

  A 3D vector of floats.

- **typedef Vector3< int > Vector3i**

  A 3D vector of int.

- **template<typename T> hum::Vector3< T > operator- (const hum::Vector3< T >&right)**

  Overload of unary negative operator.

- **template<typename T> hum::Vector3< T > & operator+= (hum::Vector3< T > &left, const hum::Vector3< T >&right)**

  Overload of binary plus-equal operator.

- **template<typename T> hum::Vector3< T > & operator-= (hum::Vector3< T > &left, const hum::Vector3< T >&right)**

  Overload of binary minus-equal operator.

- **template<typename T> hum::Vector3< T > operator+ (const hum::Vector3< T > &left, const hum::Vector3< T >&right)**

  Overload of binary plus operator.
6.28 hum::Vector3< T > Class Template Reference

- template<typename T >
  hum::Vector3< T > operator- (const hum::Vector3< T > &left, const hum::Vector3< T > &right)
  
  Overload of binary minus operator.

- template<typename T >
  hum::Vector3< T > & operator+= (hum::Vector3< T > &left, T right)
  
  Overload of binary multiply-equal operator.

- template<typename T >
  hum::Vector3< T > & operator/= (hum::Vector3< T > &left, T right)
  
  Overload of binary divide-equal operator.

- template<typename T >
  hum::Vector3< T > operator* (const hum::Vector3< T > &left, const hum::Vector3< T > &right)
  
  Overload of binary multiplication operator.

- template<typename T >
  bool operator== (const hum::Vector3< T > &left, const hum::Vector3< T > &right)
  
  Overload of binary different operator.

- template<typename T >
  bool operator!= (const hum::Vector3< T > &left, const hum::Vector3< T > &right)
  
  Overload of binary different operator.

6.28.1 Detailed Description

template<typename T >
class hum::Vector3< T >

Class representing a 3D vector.

hum::Vector3 is a simple class that defines a mathematical vector with two coordinates (x, y and z). It can be used to represent anything that has two dimensions: a size, a point, a velocity, etc.

The template parameter T is the type of the coordinates. It can be any type that supports arithmetic operations (+, -, /, *) and comparisons (==, !), for example int or float.

You generally don’t have to care about the templated form (hum::Vector3< T >), the most common specializations have special typedefs:

- hum::Vector3< float > is hum::Vector3f
- hum::Vector3< int > is hum::Vector3i

The hum::Vector3 class has a small and simple interface, its x, y and z members can be accessed directly (there are no accessors like setX(), getX()) and it contains no mathematical function like dot product, cross product, length, etc.

Usage example:

hum::Vector3f v1(16.5f, 24.f, 13.f);
v1.x = 18.2f;
float y = v1.y;

hum::Vector3f v2 = v1 + 5.f;
hum::Vector3f v3;
v3 = v1 - v2;
bool different = (v2 != v3);

Note: for 2-dimensional vectors, see hum::Vector2
6.28.2 Constructor & Destructor Documentation

6.28.2.1 template <typename T > template<typename U > hum::Vector3<T>::Vector3 ( const Vector3<U> & v ) [inline]

Cast constructor from other Vector3 of different type.

It will work always that there is a defined cast from U to T.

6.28.3 Member Function Documentation

6.28.3.1 template <typename T > T& hum::Vector3<T>::operator[] ( unsigned int position ) [inline]

Indexed accessor for the components of the vector.

Gets the corresponding component as if the vector was an array like [x, y].

Returns
   The corresponding component.

6.28.3.2 template <typename T > const T& hum::Vector3<T>::operator[] ( unsigned int position ) const [inline]

Indexed accessor for the components of the vector.

Gets the corresponding component as if the vector was an array like [x, y].

Returns
   The corresponding component.

6.28.4 Friends And Related Function Documentation

6.28.4.1 template <typename T > bool operator!= ( const hum::Vector3<T> & left, const hum::Vector3<T> & right ) [related]

Overload of binary different operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left vector.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right vector.</td>
</tr>
</tbody>
</table>

Returns
   Whether there is a difference between left and right's components.
6.28.4.2 template<typename T> hum::Vector3<T> operator* ( const hum::Vector3<T> & left, T right )
[related]
Overload of binary multiplication operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left factor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right factor.</td>
</tr>
</tbody>
</table>

Returns

The result of multiplying both factors, component by component.

6.28.4.3 template<typename T> hum::Vector3<T> operator* ( T left, const hum::Vector3<T> & right )
[related]
Overload of binary multiplication operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left factor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right factor.</td>
</tr>
</tbody>
</table>

Returns

The result of multiplying both factors, component by component.

6.28.4.4 template<typename T> hum::Vector3<T> & operator*= ( hum::Vector3<T> & left, T right )
[related]
Overload of binary multiply-equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left factor and receiver of the result.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right factor.</td>
</tr>
</tbody>
</table>

Returns

A reference to left after having multiplied both factors, component by component, and assigned the result to left.

6.28.4.5 template<typename T> hum::Vector3<T> operator+ ( const hum::Vector3<T> & left, const hum::Vector3<T> & right )
[related]
Overload of binary plus operator.
Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left addend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right addend</td>
</tr>
</tbody>
</table>

Returns

The result of adding both addends, component by component.

6.28.4.6  template<typename T> hum::Vector3<T> & operator+= ( hum::Vector3<T> & left, const hum::Vector3<T> & right ) [related]

Overload of binary plus-equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Vector that will be added the value of right.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Value to be added to left.</td>
</tr>
</tbody>
</table>

Returns

A reference to left after having added right to it.

6.28.4.7  template<typename T> hum::Vector3<T> operator- ( const hum::Vector3<T> & right ) [related]

Overload of unary negative operator.

Parameters

| right | Vector to negate. |

Returns

A copy of the input vector with all its components negated.

6.28.4.8  template<typename T> hum::Vector3<T> operator- ( const hum::Vector3<T> & left, const hum::Vector3<T> & right ) [related]

Overload of binary minus operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Left addend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Right addend</td>
</tr>
</tbody>
</table>

Generated by Doxygen
Returns

The result of substracting both addends, component by component.

6.28.4.9 template<typename T> hum::Vector3<T> & operator-= ( hum::Vector3<T> & left, const hum::Vector3<T> & right ) [related]

Overload of binary minus-equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Vector that will be substracted the value of right.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Value to be substracted to left.</td>
</tr>
</tbody>
</table>

Returns

A reference to left after having substracted right to it.

6.28.4.10 template<typename T> hum::Vector3<T> operator/ ( const hum::Vector3<T> & left, T right ) [related]

Overload of binary multiplication operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Dividend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Divisor.</td>
</tr>
</tbody>
</table>

Returns

The quotient of the division, component by component.

6.28.4.11 template<typename T> hum::Vector3<T> & operator/= ( hum::Vector3<T> & left, T right ) [related]

Overload of binary divide-equal operator.

Parameters

<table>
<thead>
<tr>
<th>left</th>
<th>Dividend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Divisor.</td>
</tr>
</tbody>
</table>

Returns

A reference to left after having been divided, component by component, by right.
Overload of binary equal operator.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Left vector.</td>
</tr>
<tr>
<td>right</td>
<td>Right vector.</td>
</tr>
</tbody>
</table>

Returns

Whether left and right are equal, component by component.

The documentation for this class was generated from the following file:

- hummingbird/include/hummingbird/Vector3.hpp
Index

～Game
   hum::Game, 35
actor
   hum::Behavior, 24
actors
   hum::Game, 35
addBehavior
   hum::Actor, 17
addPlugin
   hum::Game, 35
addShader
   mogl::ShaderProgram, 57
AnimatedSprite
   mogl::AnimatedSprite, 22
asMicroseconds
   hum::Time, 71
asMilliseconds
   hum::Time, 71
asNanoseconds
   hum::Time, 71
asSeconds
   hum::Time, 71
assert_msg
   hum, 10
bindFragmentOutput
   mogl::ShaderProgram, 57
bindVertexAttribute
   mogl::ShaderProgram, 57
Camera
   mogl::Camera, 27
clearSounds
   mogl::SoundManager, 62
deltaTime
   hum::Game, 36
disable
   mogl::Drawable, 31
draw
   mogl::AnimatedSprite, 22
   mogl::Drawable, 31
   mogl::Sprite, 66
enable
   mogl::Drawable, 31
fixedUpdate
   hum::Actor, 17
   hum::Behavior, 24
   fixedUpdateLag
   hum::Game, 36
   fixedUpdateTime
   hum::Game, 36
Game
   hum::Game, 35
game
   hum::Actor, 17
   hum::Plugin, 49
get
   mogl::SoundManager, 62, 63
getBehavior
   hum::Actor, 17, 18
getBehaviors
   hum::Actor, 18
getPlugin
   hum::Game, 36, 37
getTime
   hum::Clock, 28
handleEvent
   mogl::InputHandler, 39
hum, 9
   assert_msg, 10
   log, 10
   log_d, 10
   log_e, 11
hum::Actor, 15
   addBehavior, 17
   fixedUpdate, 17
   game, 17
   getBehavior, 17, 18
   getBehaviors, 18
   isActive, 19
   onDestroy, 19
   preUpdate, 19
   transform, 19
hum::Behavior, 23
   actor, 24
   fixedUpdate, 24
   init, 25
   onActivate, 25
hum::Clock, 28
   getTime, 28
   reset, 28
hum::Game, 33
   ~Game, 35
   actors, 35
   addPlugin, 35
mogl::MusicManager, 46
loadResource, 47
mogl::Rectangle, 50
init, 52
setShaderProgram, 52
mogl::Shader, 54
isCompiled, 55
loadFromFile, 55
loadFromSource, 55
mogl::ShaderProgram, 55
addShader, 57
bindFragmentOutput, 57
bindVertexAttribute, 57
isLinked, 57
link, 58
setUniform2f, 58
setUniform3f, 58
setUniform4f, 58
setUniformMatrix4f, 58
use, 59
mogl::ShaderProgramDef, 13
mogl::ShaderProgramManager, 59
mogl::SoundManager, 61
clearSounds, 62
get, 62, 63
loadResource, 63
play, 63
mogl::Sprite, 64
draw, 66
init, 66
setShaderProgram, 66
setTextureRect, 66
Sprite, 66
mogl::SpriteAnimation, 13
mogl::SpriteAnimationManager, 67
mogl::TextureManager, 68
loadResource, 69
MultimediaOGL
mogl::MultimediaOGL, 45
onActivate
hum::Behavior, 25
mogl::Drawable, 32
onDestroy
hum::Actor, 19
operator=
hum::Vector2, 76
hum::Vector3, 82
operator+=
hum::Vector2, 76
hum::Vector3, 82, 83
operator+=
hum::Vector2, 77
hum::Vector3, 83
operator+=
hum::Vector2, 77
hum::Vector3, 83
operator+=
hum::Vector2, 77
operator-=
hum::Vector3, 84
operator-=
hum::Vector2, 78
hum::Vector3, 84
operator/= hum::Vector2, 78
hum::Vector3, 85
operator/= hum::Vector2, 79
hum::Vector3, 85
operator==
hum::Vector2, 79
hum::Vector3, 85
operator[]
hum::Vector2, 75
hum::Vector3, 82
play
mogl::SoundManager, 63
preUpdate
hum::Actor, 19
reset
hum::Clock, 28
run
hum::Game, 37
setCamera
mogl::MultimediaOGL, 45
setDrawSpaceTransform
mogl::MultimediaOGL, 45
setOrthogonal
mogl::Camera, 27
setPerspective
mogl::Camera, 27
setShaderProgram
mogl::Drawable, 32
mogl::Rectangle, 32
mogl::Sprite, 66
setSpriteAnimation
mogl::AnimatedSprite, 22
setTextureRect
mogl::Sprite, 66
setUniform2f
mogl::ShaderProgram, 58
setUniform3f
mogl::ShaderProgram, 58
setUniform4f
mogl::ShaderProgram, 58
setUniformMatrix4f
mogl::ShaderProgram, 58
SpaceTransformation
mogl, 13
Sprite
mogl::Sprite, 66
transform
  hum::Actor, 19
  hum::Transformation, 73
  mogl::Drawable, 32, 33
Transformation
  hum::Transformation, 72
update
  mogl::InputHandler, 39
use
  mogl::ShaderProgram, 59
Vector2
  hum::Vector2, 75
Vector3
  hum::Vector3, 82