Molecular Visualization and 2D Interaction in Virtual Reality

Author:

Konstantinos Kazatzis

Supervisor:

Pere-Pau Vázquez
ViRVIG Group
Universitat Politècnica de Catalunya

A master's thesis on Computer Graphics and Virtual Reality submitted to:

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

25/6/2020

.

Abstract

Efficient molecular visualization in a virtual reality environment, requires fast and high-quality rendering of geometrical objects, as well as intuitive interaction and information visualization. Furthermore, it should also take into consideration, the challenges posed by the limitations of virtual reality, which are caused by world space occlusion, and lack of traditional interaction and selection tools. This work aims to contribute in these areas, by providing ray-casting based rendering of impostor geometry objects inside Unity, a 2D navigation and selection technique without the need to use controller-based ray casting, and an occlusion-free world space interaction panel.

In this work, we will render high-quality sphere and cylinder impostor objects through ray casting, to be used as the basis for the space-filling and ball-and-stick visualization of molecular models. These impostor objects are built inside the Unity game engine, compatible with Unity's pipeline, lighting and post-processing effects. We also implement a selection technique that allows the user to select and jump to neighboring objects without the need to ray cast into them, that facilitates selection in cluttered regions. With this technique we will visualize atom distances, bond angles, and torsion angles. Finally, we implement a world space interaction panel, that disregards occlusion to ease interaction. All of the above are concluded in a Unity powered application for molecular visualization and interaction in virtual reality.

Table of Contents

Table of Figures

List of Abbreviations

VR	Virtual Reality
HMD	Head Mounted Display
PDB	Protein Data Bank
GPU	Graphics Processing Unit
CPU	Central Processing Unit
AO	Ambient Occlusion
XR	Unity Virtual Reality modes

1. Introduction

Molecular visualization is the discipline that uses visualization techniques to represent molecular models, and facilitate interaction and information extraction from the elements that construct the model, for example, atoms, bonds and secondary structures, in order to help biologists get a deeper understanding of these elements. To represent a molecular model, one can choose one of several visualization methods, like the space-filling model, or the ball-and-stick model, that allow atom and bond level interaction, or one can provide a more abstract representation by re-constructing the surface of the secondary structures, such as alpha helixes and beta sheets. Furthermore, modern 3D visualization powered by computer graphics technology, can provide a more faithful representation when paired with realistic and real-time rendering, to ease interaction.

Towards increasing the fidelity of the visualization, Virtual Reality goes one step further, by providing stereoscopic 3D rendering and high field of view displays, that allow the exploration of much larger models. However, the current technology of VR devices, and more specifically head mounted displays, usually offer lower resolution displays compared to a common monitor display, which can be the cause of several problems when it comes to interacting and visualizing information. Furthermore, the user lacks the traditional interaction tools, the keyboard and mouse, which makes the interaction with the scene more complex.

1.1 Motivation and addressed problems

One of the drawbacks of molecular visualization in VR, is that VR requires fast and realistic rendering. Common VR devices are Head Mounted Displays which consist of one screen per eye with usually lower resolution compared to common monitor displays, but with much higher refresh rate. Refresh rate is a very crucial parameter in VR, since a 3D visualization with low refresh rate can be the cause of many problems, for example, motion sickness. The standard for VR rendering is around 90 frames per second refresh rate per eye, whereas a common monitor display can get away with less than 60 frames per second.

Another drawback of molecular visualization in VR, is that the user does not have the traditional interaction tools that is accustomed to use, the keyboard and mouse, but rather uses custom hand-held controller devices unique to each HMD brand, that usually work with ray casting or hand-extension techniques. Such techniques can be very inefficient when it comes to selecting small objects in a complex scene.

Furthermore, a common problem in VR environments, is information visualization, and world interaction. In a desktop monitor display, it is relatively straightforward to create a screen-space panel, with screen space buttons and text elements, to store information and change parameters of the visualization using the mouse. However, VR scenes do not have screen space elements, and the hand-held controllers used can only have up to a certain number of buttons. As a result, VR scenes tend to have world space panels, with virtual buttons and text, placed within the scene next to the model.

One of the problems with world space information and interaction panels, is occlusion. A world space element can be easily hidden, depending on the position of the model and the user. Furthermore, the user must also be able to interact with the panel at any time, in order to change parameters of the visualization. In VR, this problem is amplified, due to lower resolution displays.

1.2 Previous work

There has already been significant research in the area of 3D selection inside a Virtual Reality environment. Several proposals have been made towards enhancing ray casting to ease the selection of small and occluded objects, like the proposal to replace the selection ray with a selection cone with a variable apex angle [1], or to incorporate a two handed technique to define the pointing direction [2].

Other researchers have proposed eye-rooted techniques, meaning, ray casting is enhanced by using a ray that originates from the user's position and viewing angle. Such techniques use a combination of rays, to facilitate selection and visual feedback about the selected object [3]. Such techniques also combine a variety of methods, for example, the cone selection with variable apex angle.

Most recently, there has been the work of F. Argelaguet and C. Andujar [4], who used a combination of eye-rooted and hand-rooted techniques, with an eye-rooted ray for selection, and a hand-rooted ray for visual feedback, to facilitate the selection of occluded objects, and solve the occlusion mismatch between the set of visible objects between the eye and the hand.

While the above techniques significantly enhance simple ray casting, they are not suitable for such cluttered scenes as the ones with large molecules. As a result, we have investigated new interaction techniques to facilitate accurate selection in those environments that use a pointer for the initial selection, and the navigation pad for fine selection.

1.3 Contributions

The contributions of this work come in three areas. To render the scene, we are going to use Unity, which is a very common end-to-end tool, mainly used for creating video games. However, Unity provides excellent support for all VR devices out of the box, has a wide range of tools and assets offered to ease development, and already implements the "heavy lifting" of a rendering pipeline, while at the same time offering the ability for custom shaders, lightning and post processing effects. Towards that, we are going to implement Unity shaders for realistic and fast rendering of geometry objects, impostor spheres and impostor cylinders, fully integrated within Unity's lightning calculations and factory post processing effects, ready to be instantiated in a Unity scene. This part, as well as all other parts of the rendering pipeline, are covered in the chapter Rendering.

Next, we are going to implement a novel selection technique, that allows the user to move around the scene, and select neighboring objects, without the need to continuously use a 3D selection technique, but rather utilizes 2D input. Then, we are going to illustrate the effectiveness of this technique by visualizing distances between atoms, angles formed between bonds, and torsion angles formed between atoms. This part is covered in the chapter Interaction.

The last contribution is a simple approach to a world space panel, that facilitates interaction, even when the panel is occluded behind elements in the scene. This world space panel will use virtual buttons with images rather than text, and will allow the user to bring it forward for interaction, ignoring everything else in the world. This part is covered in the section Interaction: World space widget.

In the last chapter of the work, Usability experiments, we will explore possible experiments that involve real users, that could be carried out in order to validate the usability of the system.

1.4 COVID-19

Given the unique circumstances created during the 2019-2 semester, this last chapter, as well as all other sub-chapters that involve equipment that was not available, for example, the HTC Vive VR headset, or a VR capable computer, are focused mainly on how we could plan to use such equipment and how we could design and conduct such experiments. Everything is implemented with VR in mind, from rendering to interaction, by adapting the solutions as realistically as possible to a VR environment.

2. Visualizing molecular models

Before diving into the details of this work, it is necessary to provide some theoretical background regarding visualizing molecular models, as well as some biological background regarding the visualization methods that will be used, the space-filling and the ball-and-stick models, and the information that is going to be extracted from these models, which is, residues and chains, atom types, atom distances, bond angles, and torsion angles,

The molecular models that we are going to use can be downloaded from the Protein Data Bank. Our application can read files in the *.pdb* format, downloaded from the Protein Data Bank website, and extract and visualize the atoms in these models. The visualization models that are going to be used are, the Space-filling and the Ball-and-stick model. For performance comparisons, the following *PDB* models will be used in the upcoming chapters:

- 1TES: Oxygen binding muscle protein, 1435 atoms
- 4F0H: Unactivated rubisco with Oxygen bound, 5030 atoms
- 1S5L: Architecture of photosynthetic evolving center, 45945 atoms

The PDB file format stores the information in Angstrom, whereas all visualization and interaction will be done in nanometers. As a result, every value from now on without specified unit of measurement, as well as all numbers visualized, will be in nanometers.

2.1 Space filling model

The space filling model is a 3D visualization model that represents each atom as a sphere, whose radius is proportional to the radius of the element that the atom represents. The atomic radius can be calculated in many different ways, for example, using the covalent radius, or the Van der waals radius. The color of the sphere is usually the color that corresponds to the element based on the CPK coloring table.

CPK coloring table used:

Element	**RGB color**	**Element**	**RGB Color**
Hydrogen (H)	(1,1,1)	Phosphorus (P)	(1, 0.53, 0)
Carbon (C)	(0,0,0)	Sulfur (S)	(1, 0.92, 0)
Nitrogen(N)	(0,0,1)	Boron (B)	(1, 0.85, 0.6)
Oxygen(O)	(1,0,0)	Alkali Metals (Li, Na, K, Rb, Cs, Fr)	(0.54, 0.16, 0.88)
Fluorine(F), Chlorine (Cl)	(0,1,0)	Alkaline Earth Metals (Be, Mg, Ca, Sr, Ba, Ra)	(0, 0.39, 0)
Bromine (Br)	(0.54, 0, 1)	Titanium (Ti)	(0.56, 0.56, 0.56)
Iodine (I)	(0.58, 0, 0.82)	Iron (Fe)	(1, 0.54, 0)
Noble Gases (He, Ne, Ar, Xe, Kr)	(0,1,1)	Other	(1, 0, 0.8)

Covalent Radius

The covalent radius is the nominal radius of the atoms of an element when covalently bound to other atoms, as deduced from the separation between the atomic nuclei in molecules. In principle, the distance between two atoms that are bound to each other in a molecule (the length of that covalent bond) should equal the sum of their covalent radii [5]. By using the covalent radius, atoms that form a bond, barely touch one another, which reduces the complexity of the scene.

Van der Waals radius

The Van der Waals radius is half the minimum distance between the nuclei of two atoms of the element that are not bound to the same molecule [5]. The Van der Waals radius tends to be higher than the covalent, which can significantly increase the complexity in the scene. In the images below we can see the *1tes* molecular model from the same camera view, for different atomic radiuses.

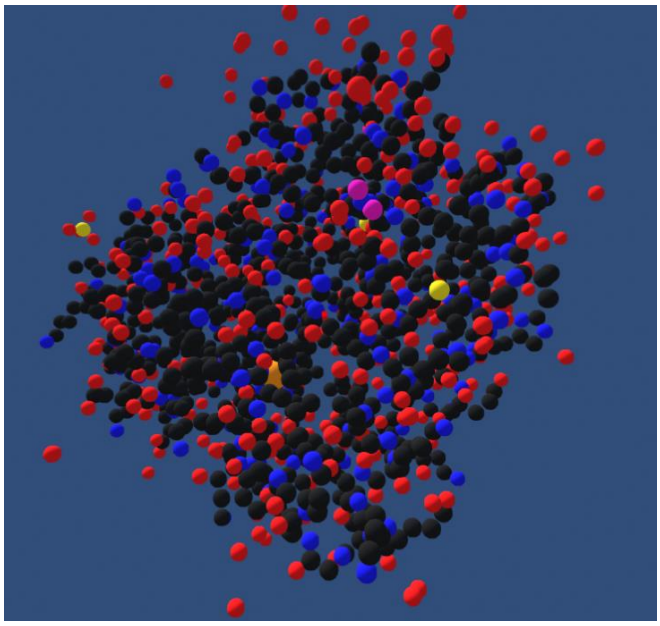


Figure 1: 1tes molecular model with CPK coloring, covalent radius and no post processing

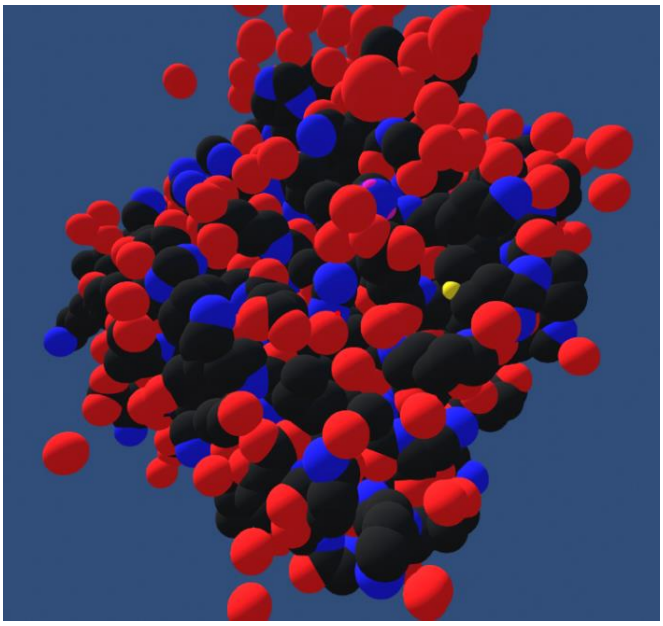


Figure 2: 1tes molecular model with CPK coloring, van der waals radius and no post processing

2.2 Ball-and-stick model

The Ball-and-stick model, is a molecular visualization model where atoms are typically represented as spheres with the same radius, connected by rods that represent the bonds between atoms. Different types of bonds are usually represented either with different rod colors, or multiple rods for double and triple bond connections. In this model, the angles between the rods should represent the angles between the atomic bonds. The color of the spheres is usually mapped to the CPK coloring scheme, just like in the Space-filling model. In Ball-and-stick model, the radius of the sphere is usually not mapped to any physical property but it is rather a radius that is much smaller than the length of the bond, which allows us to inspect the bonds and the atoms more clearly.

A *.pdb* file does not contain information about the bonds between atoms, so, in order to reconstruct the bonds, it is assumed that a connection between two atoms in a residue exists, if and only if, the atom to atom distance is smaller than the sum of their covalent radiuses. However, the covalent radius for each element cannot be measured with complete accuracy, so, a threshold of 15nm is used which is calculated empirically. This will allow the creation of bonds between the atoms, but will not recreate the type of bond. With this threshold, the total number of bonds created is:

	1TES	**4F0H**	**1S5L**
Atoms	1231	5030	45945
Bonds	1092	3895	42644

The bonds are orientated so that their length is equal to the distance between the atoms, and their radius is constant.

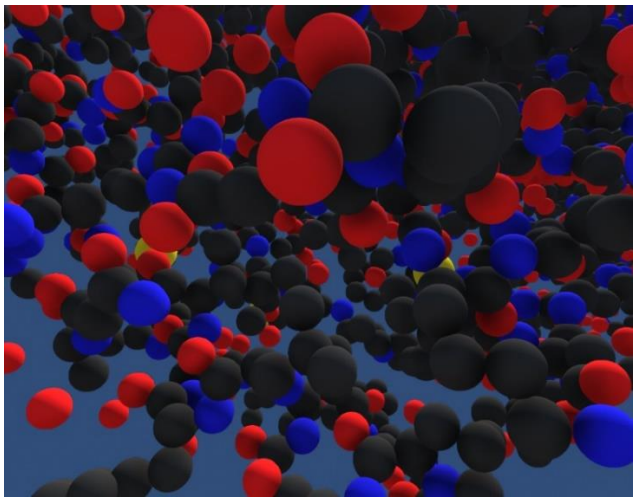


Figure 3: 4f0h molecular model, space filling model

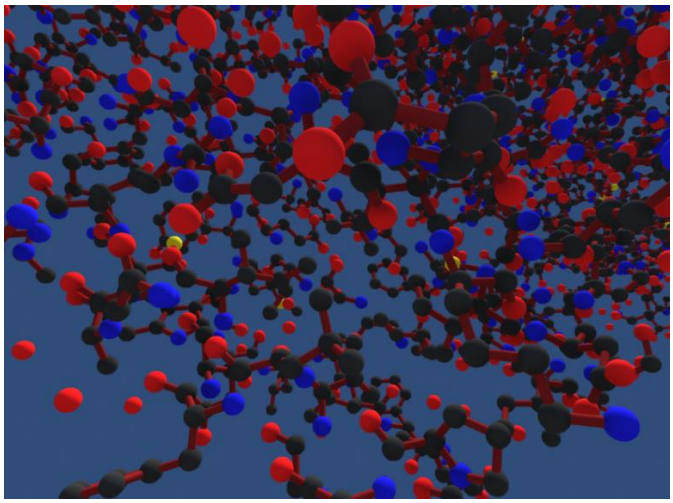


Figure 4: 4f0h molecular model, ball-and-stick

Apart from the reduced complexity in the scene, another advantage that the ball-and-stick model has, is that it gives a much better depth perspective. This is even more important in a scene that the user can manipulate and interact with.

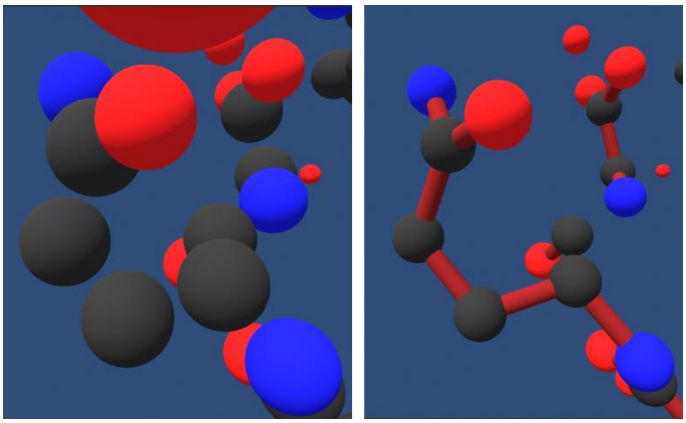


Figure 5: 1tes molecular model, space filling (left) and ball-and-stick (right), depth cue comparison

On the other hand, the Space filling model gives a better perspective of the total volume occupied. In our application, the user is able to change this parameter in real-time, and inspect the scene in any of the two visualization models.

2.3 Residues and chains

A residue refers to an atom or a group of atoms that forms part of a molecule, such as a methyl group. In biochemistry, a residue may also refer to a single molecular unity within a polymer, and residue is thus another term for monomer [6].

Chains are the individual polymers (macromolecules) contained in a PDB structure. Macromolecules are large molecules composed of thousands of covalently connected atoms. Macromolecules are formed by many monomers linking together, forming a polymer.

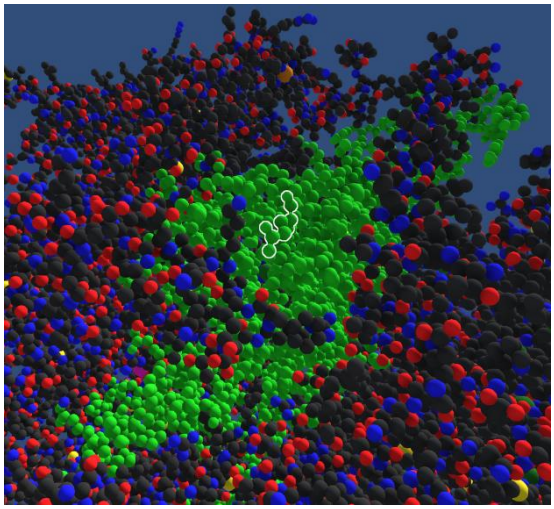


Figure 6: Visualization of a residue (highlighted) and a chain (colored green)

2.4 Model Information visualized

To examine the effectiveness of the selection techniques created, the user can use any visualization method between the space-filling and the ball-and-stick, and extract atom distances, bond angles, and torsion angles. The atom distance is the distance between the nuclei of the atoms. This value is visualized in nm. The bond angle refers to the 3D angle formed between two bonds that have a common atom. Of course, in the space-filling model where there are no bonds, this interaction is not possible. The user is also able to calculate and visualize the torsion angle between any four atoms in the scene. The torsion angle is defined as the angle between the two planes that are defined by four non colinear atoms.

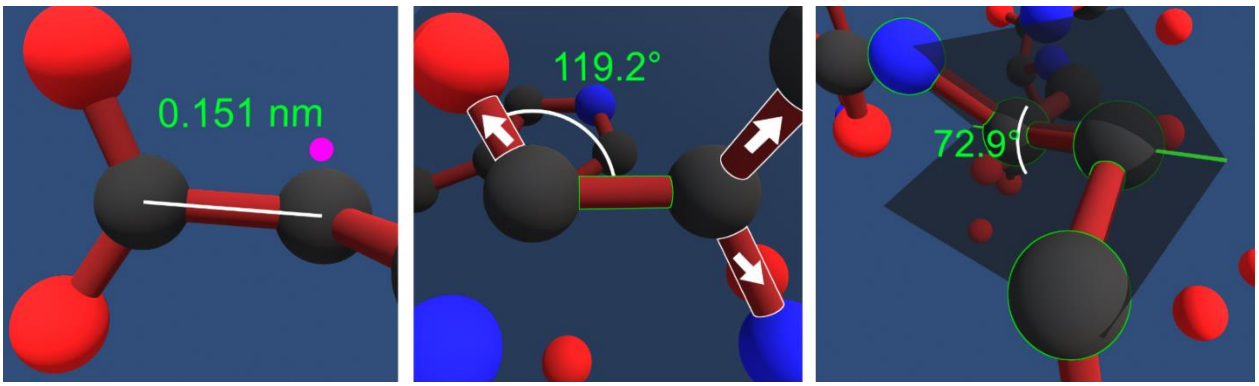


Figure 7: Visualization of atom distances, bond angles, and torsion angles

3. Rendering

To render the molecular scene, we need several things. First of all, we need to be able to effectively render high quality spherical geometry, since atoms will be represented as spheres, and cylindrical geometry, since bonds between atoms will be represented as cylinders. Rendering regular triangle meshes for spheres and cylinders, would be very inefficient for large models. As a result, we will employ a technique called impostors to generate spherical and cylindrical geometry. Next, we will use deferred shading so that we can apply post processing effects, but also take advantage of Unity's lightning pipeline. That means that the final color calculation will be done by Unity's physically based rendering pipeline. Furthermore, to avoid having thousands of rendering calls, one for each object in the scene, we will use a technique called instancing, which allows us to group together GPU draw calls for objects that have the same geometry. Then, we will use post processing effects like Ambient Occlusion and sphere highlighting to increase the realism of the scene, and give visual feedback for user interaction. So, the full rendering pipeline will be, sphere and cylinder impostors to draw atoms and bonds, instancing to batch draw calls together, physically based rendering for lightning and post processing effects.

The rendering techniques above are by no means revolutionary. So, the next chapters are devoted onto how they are implemented within the Unity environment.

3.1 Sphere Impostors

Rendering a high-quality sphere through the common screen rendering pipeline can be very expensive, especially if we want to render thousands of spheres. The reason is that, it is not possible to represent continuous geometry, for example curves, while using the common rendering pipeline of vertices and triangles. Furthermore, significantly increasing the number of vertices/triangles, to faithfully represent a curved surface is not a viable solution, since it is going to significantly worsen the performance. To demonstrate this, we can continuously subdivide a sphere from a cube, and we will see that in order to reach an acceptable looking sphere, we have to reach a very high number of faces.

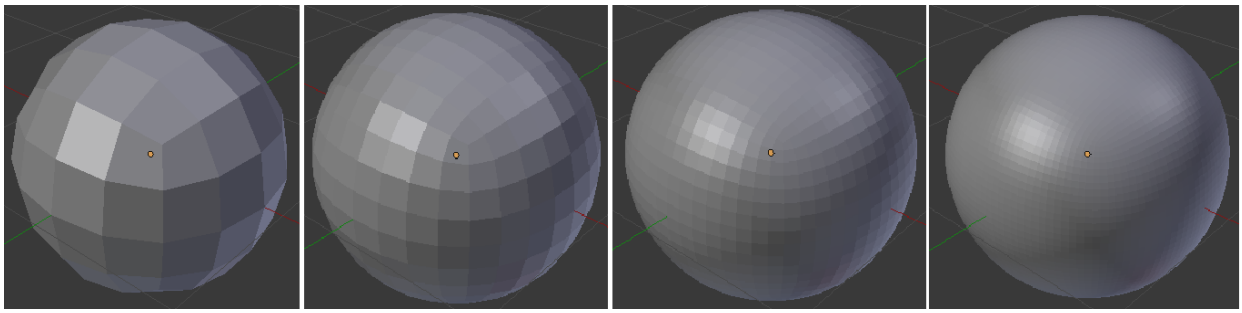


Figure 8: Cube subdivided spheres

A common technique widely known as sphere impostor, is to render a simple camera-oriented quad, and generate the sphere geometry in the fragment shader. The idea is that, the camera-oriented quad will generate the fragments necessary to draw a sphere on the screen, if we were to map the sphere from the world to the screen. Then in the fragment shader, for each fragment, the actual fragment position is calculated by doing ray casting from the camera to the fragment and calculating the intersection of the ray with the desired geometrical sphere. This can effectively increase the quality of the curve of the sphere to basically the resolution of the screen, while maintaining the number of rendered vertices to a simple quad geometry.

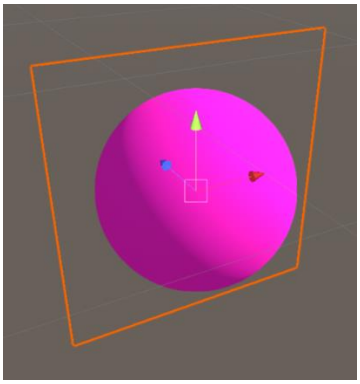


Figure 9: Sphere impostor in Unity

To achieve this effect in Unity, we are going to define a Vertex/Fragment shader pipeline. Unity calls these shaders, *Unlit shaders*, in order to differentiate them with *Surface shaders*. *Surface shaders*, as the name suggests, are used to perform basic coloring algorithms on the surface of an object, without worrying too much about vertices and triangles and lightning. Since we want to actually change the geometry of the rendered fragments, we have to use *Unlit shaders*. Then, we are going to define a Unity Material and set that this material is implemented by our shader, and attach the material on a quad in the scene. The uniform data of the shader or the Properties of the shader as Unity calls them, like the radius and the color, will be exposed as material attributes.

Unlit Vertex Shader

We first generate a Unity Quad object. A unity quad is a quad plane that spans from (-0.5, -0.5, 0) to (0.5, 0.5, 0), facing down the *-z* axis. By passing this geometry through the vertex shader, we can use the vertices to create our camera-oriented quad in view space.

```
struct appdata {
    /* Object space position */
    float4 vertex : POSITION;
};
struct v2f {
    /* clip space position */
    float4 pos : SV_POSITION;
    /* view space position */
    float4 view_pos : TEXCOORD0;
};

v2f vert(appdata input)
{
   v2f output;

   output.view_pos = mul(UNITY_MATRIX_MV, float4(0.0, 0.0, 0.0, 1.0)) +
BOX_CORRECTION * float4(input.vertex.x, input.vertex.y, 0.0, 0.0) * 2.0f *
float4(radius, radius, 1.0, 1.0);
   output.pos = mul(UNITY_MATRIX_P, output.view_pos);

   return output;
}
```

Unity provides three possible inputs to a vertex shader, *appdata_base*, *appdata_tan*, *appdata_full* [7]. Since only the position of the object vertices is needed, we are going to use the basic input struct.

The output of the vertex shader can be any struct that contains an attribute of type *SV_POSITION*. Unity is going to use this attribute as the clip space coordinates. We are also going to use an extra attribute, the view space position, since we are going to need it in the fragment shader. To calculate the camera-oriented, view-space vertex positions, we are going to use the following formula:

$$viewSpacePos = [0,0,0,1] * M * V + [x, y, 0,0] * 2 * (r, r, 1, 1)$$

where *M = Model matrix*, *V = View matrix*, and *x, y* are the input object-space vertex coordinates. Since we render a quad from -0.5 to 0.5 in the *xy* plane, the above formula will multiply by 2 to make the vertices span from -1 to 1, multiply the result of this with the sphere radius, and then add this vertex to the view space position of the model. That way we generate a camera-oriented quad with size equal to the size of the sphere. We are also going to multiply with a *BOX_CORRECTION* factor. This factor is used to compensate for the projection of the quad to the screen.

Unlit Fragment Shader

```
fragment_output frag(v2f input) : COLOR
{

    /* Compute real fragment world position and normal */
    float3 normal_world, position_world;
    ImpostorSphere(mul(UNITY_MATRIX_I_V, input.view_pos), radius,
position_world, normal_world);

    /* Calculate depth */
    float4 clip = mul(UNITY_MATRIX_VP, float4(position_world, 1.0f));
    float z_value = clip.z / clip.w;

    /* Set depth, and calculate color */
    /* .... */
}
```

The function *ImpostorSphere*, will take as input the fragment's world position and the radius of the sphere, and will calculate the position and the normal that this fragment should have. To do that, *ImpostorSphere* will cast a ray from the camera towards the fragment, and find the closest intersection point with the perfect geometrical sphere that we want to render. If such point does not exist, it will clip the fragment using Unity's *clip(float)* method. To represent a sphere we need two attributes, a center and a radius. The *ImpostorSphere* function will assume that the sphere is placed at (0,0,0) in object space. That way the position is encoded inside the Model matrix which lowers the amount of shader properties.

The Sphere-Ray intersection algorithm is a basic sphere-line intersection algorithm that solves the second-degree equation. You can find the full code for this function in Appendix A: Ray-Sphere intersection.

By calculating the closest intersection point to the sphere, we essentially perform back-face culling for the sphere. If we want to do front-face culling, we can use the farthest intersection of the ray with the geometrical sphere.

After we calculate the world space position that we want this fragment to have, we transform it to clip space, divide with the *w* coordinate, and calculate the new depth of that fragment. We are going to give this depth value to Unity to perform depth testing.

After we calculate the geometry of the fragment and set the depth, we will proceed to calculate color and set the fragment's shader output.

Fragment shader output semantics

In order for Unity to accurately perform depth testing, we have to set the new fragment depth, which is calculated from the clip space position of the intersection point.

Unity allows the fragment shader to return multiple values, either to be used by us, or to be used by Unity, and calls these outputs *Fragment shader output semantics* [8]. One semantic that we can define, is multiple render targets *SV_TargetN*. This tool is useful when we want to render to multiple places for example textures, in order to perform custom post processing effects. Another output semantic is *SV_DEPTH*, which allows us to set the depth for that fragment. We are going to use both:

```
struct fragment_output
{
    half4 diffuse : SV_Target0;
    half4 specular : SV_Target1;
    half4 normal_world : SV_Target2;
    half4 emission : SV_Target3;
};

fragment_output frag(v2f input, out float outDepth : SV_Depth) : COLOR
{
    /* Calculate impostor*/
    /* ... */

    /* Calculate depth */
    float4 clip = mul(UNITY_MATRIX_VP, float4(position_world, 1.0f));
    float z_value = clip.z / clip.w;
    outDepth = z_value;

    /* ... */

    /* Set output parameters */
    fragment_output o;
    /* Set values to o */
    /* ... */
    return o;
}
```

All we have to do is set the calculated *z_value* to the *SV_DEPTH* variable, and then Unity will use it with whatever depth encoding formula they have to perform depth testing. We can also see the Multi-Target Rendering output that the fragment shader has, which will be explained in more depth in the Deferred Rendering section. In the following image, we can see impostor spheres in pink, and unity geometry spheres in blue:

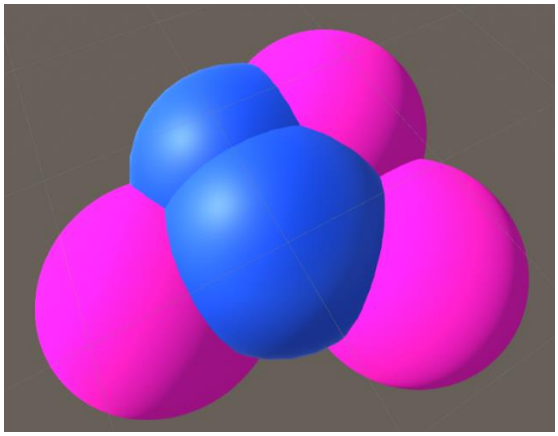


Figure 10: Real(blue) and impostor spheres(pink), depth testing

3.2 Cylinder Impostors

Now that we have seen how sphere impostors are rendered, we can generalize this approach to basically render any kind of geometrical structure, like cylinders, cones, etc. The two things that we have to change is, the initial geometry rendered that defines the fragments on the screen that will render the geometrical structure, and the ray casting algorithm that calculates the actual fragment position.

For sphere impostors, rendering a camera-oriented quad was enough, since a sphere looks the same, geometrically, from every angle. For a cylinder impostor however, this is not true. As a result, we are going to render a simple cube with size and position that tightly covers the desired cylinder object. We are going to use a single base mesh, an object-space cube that 'sits' on top of the xz plane, and has size 1 across every dimension. This is a cube that tightly covers a cylinder that sits on top of the xz plane, has radius 0.5, and height 1 with direction along the y axis. We can use this single mesh object, along with different translation, scaling and rotation matrices to place the cylinder anywhere in the scene. That way, the *Unlit Vertex Shader* is very simple, and just transforms the vertices to clip space.

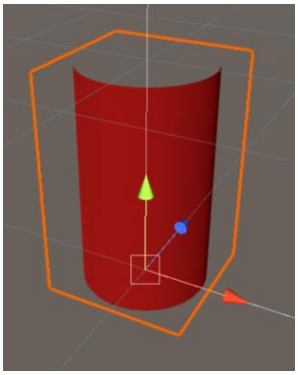


Figure 11: A cube geometry (orange) that covers an impostor cylinder

The *Unlit Fragment Shader* has to calculate the actual fragment position by performing ray casting from the camera to the fragment world position, against the desired geometrical cylinder.

In order to represent a geometrical cylinder, we define the following properties: $C \in R^3, \vec{e} \in R^3, ||\vec{e}|| = 1,\ r > 0,\ h > 0$, where C is the center of the cylinder, $\vec{e}$ defines the direction, and r and h are the width and the height. To lower the amount of properties sent to the shader, the shader assumes that C is *(0,0,0)*

in object-space, and $\vec{e}$ is the object-space *y* axis transformed to world space. We will also use the scaling that the object has in the *y* direction as the height of the cylinder. That way, the only additional property sent to the shader is the radius, since everything else is encoded within the Model matrix. Like in the sphere impostors, a correction factor must be applied in the size of cube rendered, to compensate for the projection onto the screen. This correction is applied as a scale to the *x* and *z* components of the cube

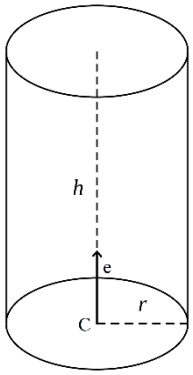
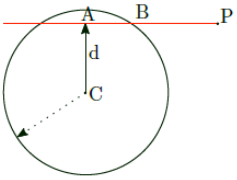


Figure 12: Cylinder representation

The Cylinder-Ray intersection algorithm implemented calculates the intersection point assuming an infinite cylinder, and does the following:



Assume that the ray is defined by the equation: $P + \lambda * \vec{v}$, and the cylinder axis is defined by the equation: $C + \mu * \vec{e}$. First, we will find the point A along the ray that is closest to the cylinder axis $\vec{e}$. To do that, assume the direction vector defined by, a random point on the ray A, and a random point on the axis M: $\overrightarrow{A-M}$, and we will solve the system: $(\overrightarrow{A-M}) * \vec{v} = 0$, $(\overrightarrow{A-M}) * \vec{e} = 0$. This system has a solution if the ray direction and the cylinder direction are not parallel. By solving the system, we get the value λ, that gives us the point A. Then, we will calculate the distance d of this point to the cylinder axis. If this distance d is greater than the radius, then there is no intersection. Otherwise, there is.

We are going to calculate the point *B*, by using the *PB* distance. The value λ that defines point *A*, is also the distance between *A* and *P*. If the plane defined by A, P, M was perpendicular to the cylinder axis, then AB distance can be found by using the Pythagorean theorem directly. Since that is not generally the case, we have to scale the AB value with the angle inclination. Thus:

$$|PB| = \lambda' = |AP| - |AB| = \lambda - \frac{\sqrt{r^2 - d^2}}{\sqrt{1 - (\vec{e} * \vec{v})^2}}$$

This λ' will give the intersection point B, $B = P + \lambda' * \vec{v}$, for the infinite cylinder. After this point is calculated, we clip it based on the cylinder height, and calculate the normal. You can find the full code for this calculation in Appendix B: Ray-Cylinder intersection.

In general, we would also want to calculate the intersection points on the top and bottom of the cylinder, assuming planes perpendicular to the cylinder direction. However, since we are going to use this cylinder as a bond between atoms, this is not needed.

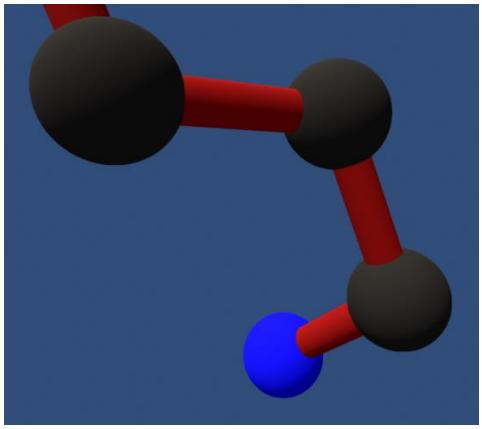


Figure 13: Impostor spheres and cylinders, no post processing

3.3 Deferred rendering

Deferred rendering or deferred shading, is a technique in which while passing through the geometry of the scene, we don't draw directly on the screen, but we rather draw/store all the necessary information onto texture buffers. These texture buffers are called the G-Buffer. The G-Buffer will hold all the data that we need in order to perform the lightning calculations, for example, position, normal, color, etc., for each fragment and in the final pass, the color for each fragment will be calculated by drawing a simple quad that covers the screen.

Deferred rendering is mainly used when we have many lights on the scene, since we can reduce the contribution of each light to a small area of screen fragments, and not for all rendered objects. Another advantage of deferred rendering, is that it allows us to perform more complicated post processing effects, that with forward rendering we couldn't. In our case, by using the deferred rendering path, we also have the advantage that we can use Unity's Physically based rendering. If we were to use forward rendering, we would have to perform lighting calculations ourselves, which can be a significant downgrade from Unity's lightning pipeline. Furthermore, if we were to use forward rendering, we would have to implement all the post processing effects ourselves, since the impostor geometry is calculated in the fragment shader. That means that if we were to use Unity's post processing pipeline, all the effects would be calculated and applied in the scene as if we are drawing quads and cubes for spheres and cylinders.

The main disadvantage of deferred rendering is that it's not possible to handle semi-transparent objects, and transparency needs to be handled with a different rendering pipeline. But since our main goal does not necessarily include any transparency, this does not have any impact on our scenes. This is not such a big drawback, since Unity allows both deferred and forward rendering in the same scene. Another disadvantage is that the G-Buffer and its size is usually a bottleneck in performance. Many graphics cards have a limited amount of render targets available, and limited memory and memory bandwidth available. However, most graphics cards have no problem running deferred rendering pipelines. Another disadvantage that comes from the limited memory of the G-Buffer is that we can't have multiple materials

on an object. The above disadvantages are not significant enough to force us to use forward rendering for our scene.

Furthermore, as mentioned in the previous paragraph, Unity allows different rendering queues based on the type of geometry rendered, and when the rendered geometry is set to transparent, then the forward rendering queue is used by default. As a result, we are also going to define forward rendering shaders that implement a simple Phong Shading model that we are going to use to render transparent spheres or cylinders.

G-Buffer content and Physically based rendering

To define that a shader is to be used in a deferred rendering pass, we apply the tag:

```
Tags { "LightMode" = "Deferred" }
```

Shader tags is a way for a shader to tell Unity how to use the shader, in terms of which rendering queue to use [9], and which lighting pass to use [10]. When a shader that unity needs does not exist, Unity is going to use the fallback shader. For example, if we were to enable shadows in our scene, this would be the result, since we haven't defined a *ShadowCaster* lightning pass, and Unity will use the default one:

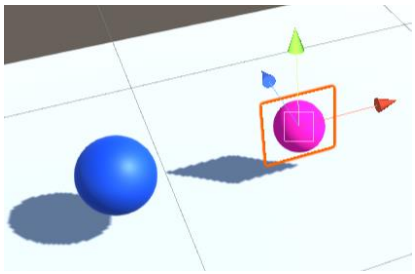


Figure 14: Impostor sphere shader with default shadows

Implementing a *ShadowCaster* pass for a sphere or cylinder impostor is straightforward, we apply the same ray casting algorithm, but from the point of view of the light, and the only output is the depth.

Let's examine now the contents of the G-Buffer [11]:

- Render target 0: ARGB32 format: Diffuse color (RGB), occlusion (A)
- Render target 1: ARGB32 format: Specular color (RGB), roughness (A)
- Render target 2: ARGB2101010 format: World space normal (RGB), unused(A)
- Render target 3: ARGB2101010 (non-HDR) or ARGBHalf (HDR) format: Emission + lightning + lightmaps + reflection probes buffer
- Depth and stencil buffer
- Render target 4: ARGB32 format: shadowmask or distance shadowmask if enabled

We have already seen the in the previous chapter that we are going to use the first four textures (depth and stencil buffer handled by Unity):

```
struct fragment_output
{
    half4 diffuse : SV_Target0;
    half4 specular : SV_Target1;
    half4 normal_world : SV_Target2;
    half4 emission : SV_Target3;
```

```
};
```

In the first texture, in the *RGB* components we store the diffuse color value of the object. In the *A* channel we store the occlusion factor. Since we are not going to use this feature, we set its value to 1. In the second render texture we store the specular color (*RGB*), and in the A channel we store the glossiness of the object. The third render texture uses 10 bits for the *RGB* channels, and 2 bits for the unused *A* channel. We store in the RGB channels the world space normal encoded as a regular normal map. In the alpha channel of this texture we still store the border highlighting information, which we are going to use for object highlighting as a Post Processing effect. In the fourth texture we will only store the ambient component of the object:

```
fragment_output frag(v2f input, out float outDepth : SV_Depth) : COLOR
{
    /* Calculate impostor geometry */
    /* Set depth */
    /* ... */

    /* Get albedo and is_highlighted values for this object */
    /* ... */

    /* Calculate diffuse and specular component from using Unity's
Physically based rendering pipeline */
    half3 specular;
    half specularMonochrome;
    half3 diffuseColor = DiffuseAndSpecularFromMetallic(albedo, metallic,
specular, specularMonochrome);

    /* Set output parameters */
    fragment_output o;
    o.diffuse = float4(diffuseColor, 1);
    o.specular = half4(specular, gloss);
    o.normal_world.xyz = normal_world * 0.5f + 0.5f;
    o.normal_world.w = is_highlighted;
    o.emission.xyz = ambient_factor * diffuseColor;
    return o;
}
```

We don't store the world space position of the fragment in the G-Buffer, since Unity will calculate that using the depth of the fragment.

By using Unity's function *DiffuseAndSpecularFromMetallic()* we can calculate the diffuse and specular components from the metallic value, and then set them in the G-Buffer textures. This will allow Unity to calculate the final color using physically based rendering in the final pass.

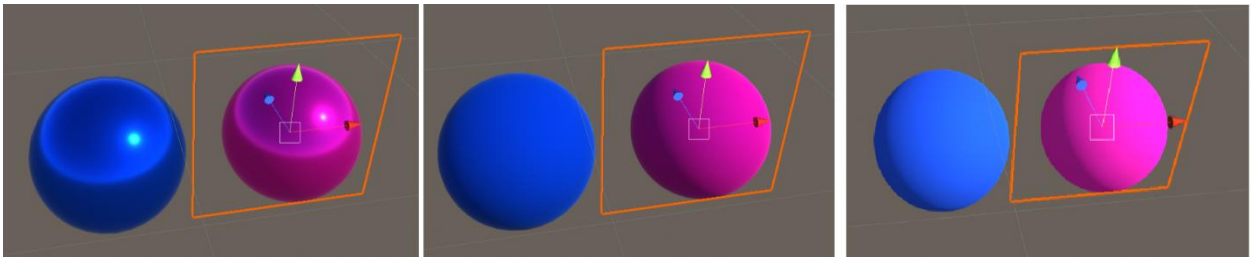


Figure 15: Physically based rendering impostor sphere(pink) and Unity sphere (blue), from left to right: glossy, metallic, our look

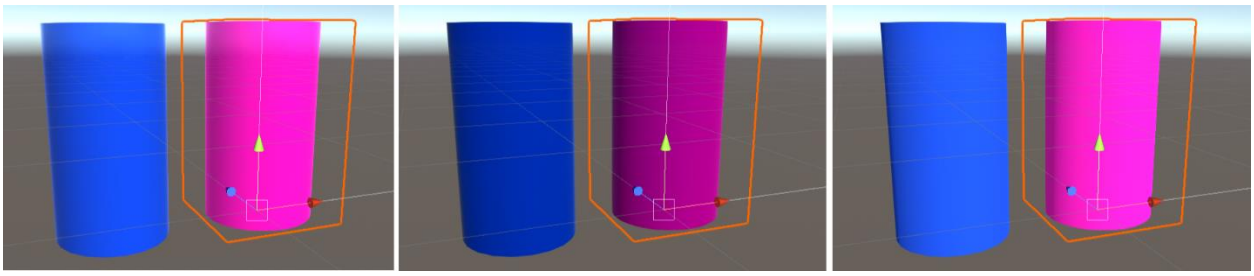


Figure 16: Physically based rendering impostor cylinder(pink) and Unity cylinder(blue), from left to right, glossy, metallic, our look

We opted for the look in the outmost right image in *Figure 15, 16*, which is zero metallic component and zero glossiness.

3.4 Instancing and material property blocks

Drawing thousands of objects in a scene can have a significant impact in performance, since the CPU must circle through all of them and draw them one by one. Unity has two ways to reduce this cost called batching and instancing. Batching, which happens automatically, either static or dynamic, groups together the meshes of all small enough objects into one huge mesh, and then draws this mesh. Static batching happens automatically to all objects marked as static, if they are small enough [12]. Dynamic batching has to be enabled. However, batching does not help us with impostors, since we actually use the vertices of each individual quad in the vertex shader to generate the camera-oriented quad. As a result, in order to reduce the number of calls we can only use instancing.

Instancing is a rendering technique that allows us to draw the same exact mesh with different parameters, using a single draw call [13]. In Unity, two objects can be rendered with a single draw call, if and only if, they have the same exact mesh, material, and material attributes as well. However, material attributes can be different in each instance, without breaking instancing, with a technique that Unity calls material property blocks, which we are going to use to set different properties, like colors, radiuses, etc. to the instantiated objects. Since both impostor spheres and impostor cylinders use the same base mesh, we can batch all impostor draw calls through instancing.

Adding instancing support

To add instancing support to the Impostor shader, we have to add a couple of things. We have to add a new preprocessor directive to the shader (*#pragma multi_compile_instancing*) to inform Unity that this shader supports instancing. Next, we have to add the instance id to the shader. This instance id will allow us to get per instance data.

```
struct appdata {
    /* Instance ID */
    UNITY_VERTEX_INPUT_INSTANCE_ID
    /* Object space position */
    float4 vertex : POSITION;
};
struct v2f {
    UNITY_VERTEX_INPUT_INSTANCE_ID
    /* clip space position */
    float4 pos : SV_POSITION;
    /* view space position */
    float4 view_pos : TEXCOORD0;
```

```
};
```

Currently, only the model matrix is different for each instantiated object, but we will add more detailed properties later. We also have to change the vertex shader to be aware of their instance id and set up the fragment shader instance id.

```
v2f vert(appdata input)
{
   v2f output;
   UNITY_SETUP_INSTANCE_ID(input);
   UNITY_TRANSFER_INSTANCE_ID(input, output);

   /* Calculate camera-oriented quad */
   /* ... */

   return output;
}
```

```
fragment_output frag(v2f input, out float outDepth : SV_Depth) : COLOR
{
    /* Set up instance id */
    UNITY_SETUP_INSTANCE_ID(input);

    /* Rest of fragment shader code */
    /* ... */

}
```

The impostor shaders now support instancing. We only have to enable it through the Unity Editor. However, all instances must have the same attributes: radius, color, etc. To change that we have to add support for material property blocks.

Per-instance data

If we were to change the properties of a material (for example color) attached to an object after instantiating the object, then Unity would duplicate the material for that object, and as a result, break batching since the new object has now a different material. Furthermore, if we were to change the shared material properties for that object, then all instances would see the same change. As a result, we have to use something that Unity calls Material Property Block.

Material property blocks are small structs that contain data, which we can set to the mesh renderer attached to the object, to override the set material properties. For sphere impostors, we only want to be able to change the color of spheres, whether or not a sphere is highlighted, and the radius of the sphere. As a result, we want 5 floating point parameters. Since material property blocks only support parameters of type *Color(float, float, float, float)*, we are going to use one parameter to store the color of the sphere to the RGB channels and the *is_highlighted* field to the alpha channel, and another parameter to store the radius, the ambient factor, and the metallic and glossiness factors, even though we are not going to use them.

For cylinder impostors we want to be able to change the color, since the color could indicate a different type of bond, the radius and the height of the cylinders, and also the highlighting information. Since the

height is encoded inside the Model matrix, we again have to use two Color properties. The rest of the attributes will be the ambient factor, metallic and glossiness.

To add support to our shader for per-instance properties, we add the following lines to the shader pass in the same visibility level with the input-output structs:

```
/* Unpack extra instance properties */
UNITY_INSTANCING_BUFFER_START(Props)
    UNITY_DEFINE_INSTANCED_PROP(float4, _Albedo)
    UNITY_DEFINE_INSTANCED_PROP(float4, _RadiusAndShading)
UNITY_INSTANCING_BUFFER_END(Props)
```

To get the properties in the vertex and fragment shader we use the following commands to get the two material block attributes, and then we unpack each channel for each property:

```
float4 radius_and_shading = UNITY_ACCESS_INSTANCED_PROP(Props,
_RadiusAndShading);
float4 albedo = UNITY_ACCESS_INSTANCED_PROP(Props, _Albedo);
```

By using Unity's Frame Debugger, we can see that we are able to batch thousands of G-Buffer draw calls down to 5, in a scene with multiple colors and radiuses:

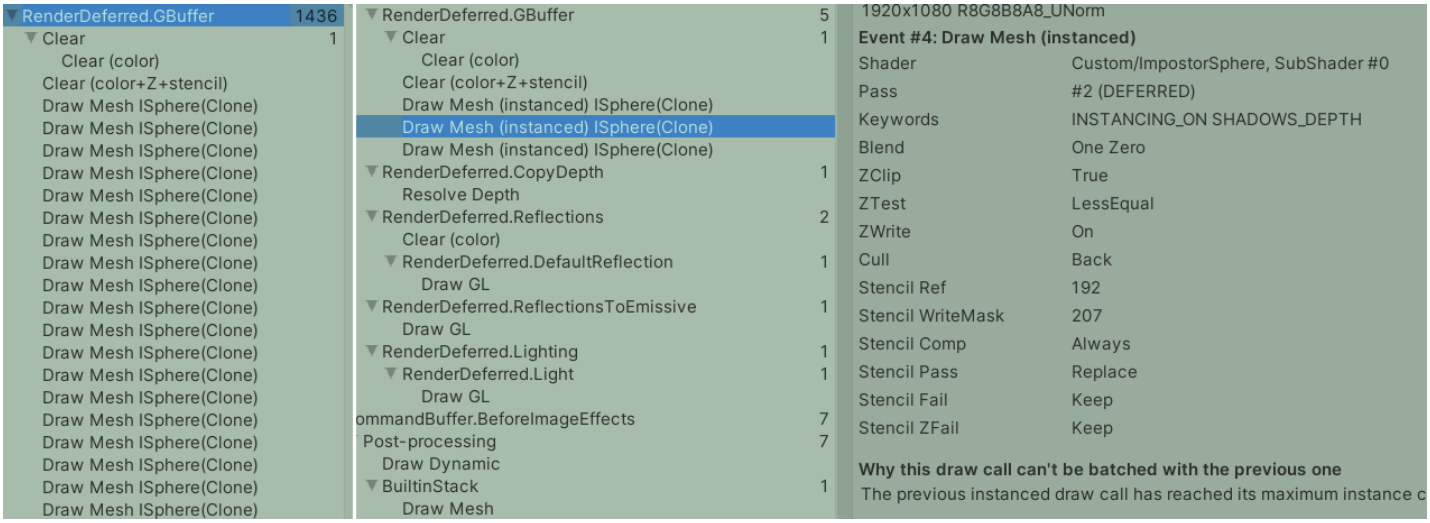


Figure 17: Left G-Buffer rendering without instancing, Right G-Buffer rendering with instancing

In *Figure 17*, in the right image we can also see why some draw calls cannot be batched with the previous ones, which is because the maximum capacity is reached. The maximum capacity of a draw call is a GPU specific parameter.

3.5 Forward Rendering pass

The forward rendering pass is identical with the deferred rendering in terms of the vertex program, and only differs in the fragment shader output. The fragment shader will unpack the properties of the instance, and calculate the final color with Phong shading, Appendix D: Phong shading functions. The forward rendering pass also defines *ForwardBase* and *ForwardAdd* passes, which are called for the directional light, and the point lights (maximum four per drawn object) in the scene, respectively.

You can find the full code of the shader for the deferred and forward pass of the sphere impostor in Appendix C: Sphere impostor Unity deferred and forward shader. The cylinder impostor is very similar.

3.5 Post processing stack

Post processing effects can significantly increase the realism of the scene without having a big impact on performance. Since our scene is now fully rendered through Unity's pipeline, we can take advantage of the already implemented post processing effects that Unity offers by default, for example Ambient Occlusion, Fog, Bloom, color filters etc., or even download from the Asset store, and we don't have to implement them from scratch. From Unity's effects, we are only going to add Ambient Occlusion to the post processing stack, and we are also going to implement a custom post processing effect for highlighting objects.

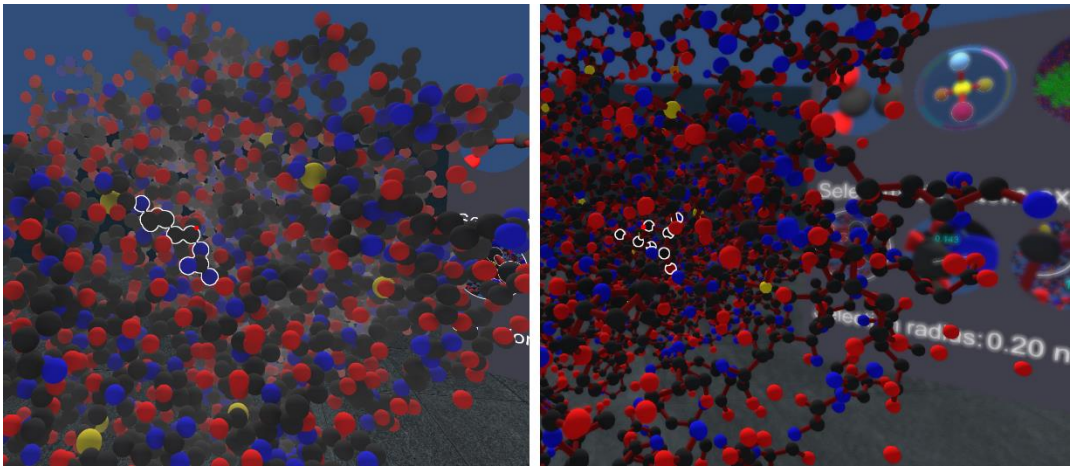


Figure 18: Fog post processing effect (left), depth of field effect (right)

Ambient Occlusion

Ambient Occlusion is a technique that tries to approximately calculate how exposed a fragment in the scene is to ambient lightning, and then apply that factor to the lightning calculations. Unity offers two Ambient Occlusion techniques: Scalable Ambient Obscurance and Multi-Scale Volumetric Obscurance/Occlusion [14]. Unity suggests that, if we target a modern Desktop platform, to use Multi-Scale Volumetric Obscurance which is a technique that utilizes compute shaders. However, low end machines may suffer from flickering using that technique. Since the current version of the system was developed on a low-end machine, the AO method used is Scalable Ambient Obscurance. This option however can be changed easily.

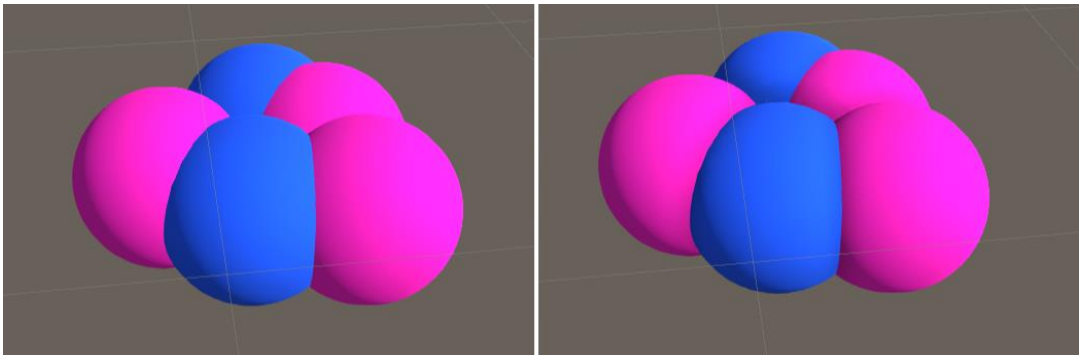


Figure 19: Ambient Occlusion with impostor (pink) and Unity spheres (blue)

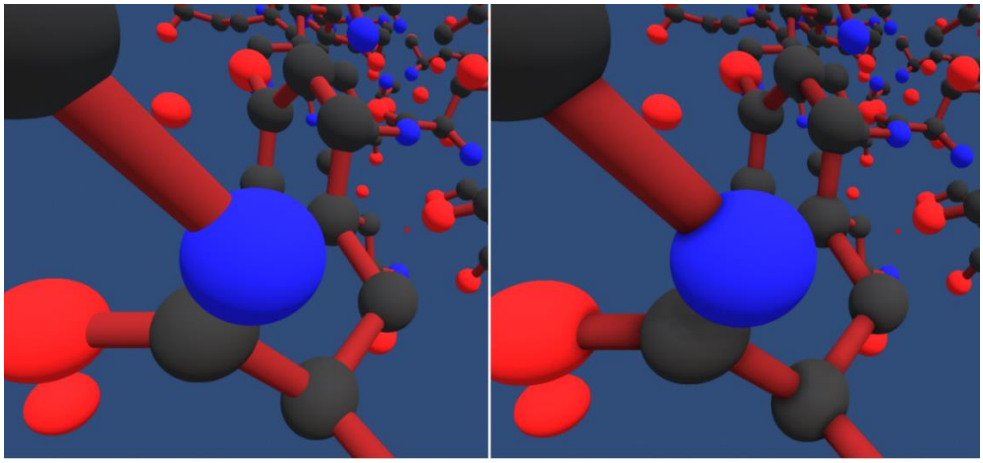


Figure 20: Ball-and-stick visualization, Multi-Scale Volumetric Obscurance AO

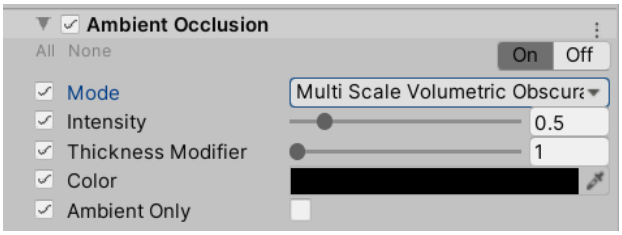


Figure 21: Ambient Occlusion parameters

Silhouette highlighting

Since we are interested into interacting with the atoms and bonds in the scene, it is crucial to give some visual feedback to the user about which atom or bond is selected, or which collection of atoms is connected into a single molecule, or a single chain. To that end, we will implement a highlighting algorithm as a post processing effect.

The algorithm that we are going to implement will get as input a texture, where the color-marked fragments correspond to objects that must be highlighted, and will calculate a border for the marked fragments. That border will then be drawn on top of the screen. We have already seen in the Deferred shading section how we mark on a texture if an object is highlighted or not. We set the alpha channel of the third render target. Since the alpha channel of Render Target 2 only has 2 bits available, we are limited to 4 different border colors, encoded into shades of grey. Thus, we are going to use: No border, white, blue and green.

If we use the Unity frame Debugger, we can examine the content of that channel in the following figures:

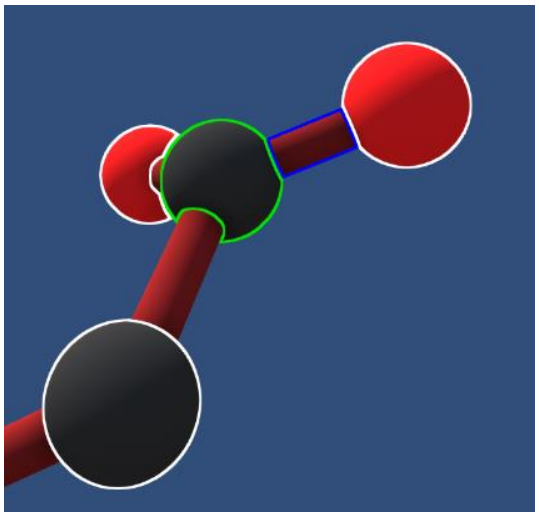


Figure 22: Highlighted group of atoms

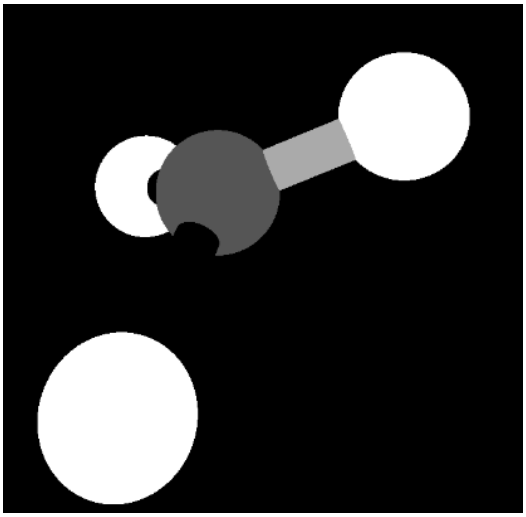


Figure 23: Alpha channel of G-Buffer Render Target 2

To create a custom post processing effect in Unity, we first have to create a shader for that effect. The idea is that, for every fragment in the highlighted texture, if it belongs to a highlighted object (its alpha channel is not 0), sample an area of fragments around it, and if any of the sampled fragments is not marked with the same color, then this fragment belongs to the highlighting border. In that case, return the color of the border. Below you can find the implementation:

```
float4 Frag(VaryingsDefault i) : SV_Target
{
    /* Get size information */
    float2 tex_size = _CameraGBufferTexture2_TexelSize.xy;
    /* Get value for current fragment */
    float selected_value = tex2D(_CameraGBufferTexture2, i.texcoord).w;
    bool is_border = false;
    /* If this fragment is highlighted, it could belong to the border */
    if (selected_value > 0) {
        /* Sample a number of fragments around, and if any of them is not
highlighted
           then this fragment belongs to the border */
        [loop]
        for (int x = -_Thickness; x <= +_Thickness; x++) {
            [loop]
            for (int y = -_Thickness; y <= +_Thickness; y++) {
                if (x == 0 && y == 0) {
                    continue;
                }
                float2 offset = float2(x, y) * tex_size;
                if (tex2D(_CameraGBufferTexture2, i.texcoord + offset).w !=
selected_value) {
                    is_border = true;
                }
            }
        }
    }

    /* If it's border return color of the outline based on the value of the
       texel*/
    if (is_border) return GetHighlightColor(selected_value);
```

```
    /* Otherwise, return previous color */
    return SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, i.texcoord);
}
```

We then create a driver script for that shader, based on Unity's post processing template, and we can register the custom effect into Unity's post processing stack. This gives us the ability to control the parameters of that shader in a single place along with all other post processing effects, and also control the order in which the post processing effects are applied:

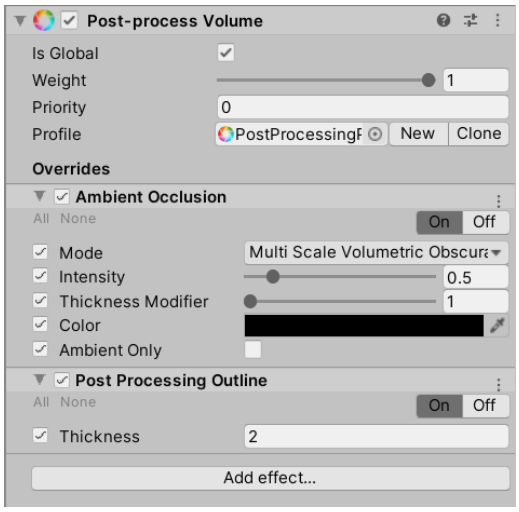


Figure 24: Post processing stack

One disadvantage that this approach has, is that we cannot have many different colors for the border. The alpha channel of render target 2 has only two bits available. That means that we can only encode four different colors. A different approach could be to pass highlighted objects a second time, and use a Unity Custom Render Texture to store the highlighting information. However, this approach adds an additional draw call for all the highlighted objects, which could worsen performance when highlighting many objects.

You can find the full shader code for this post processing in Appendix E: Border highlighting post processing effect Unity shader.

3.6 Transparency, UI and rendering order

As mentioned in previous chapters, we can combine deferred and forward rendering into a single frame, by taking advantage of Unity's rendering pipeline, and draw transparent geometry along with the user interface. As a result, it is important to give the flow of the complete rendering pipeline of the objects that can compose a single frame.

Transparent impostors

Transparent geometry in Unity, is always rendered after the opaque geometry, using a forward rendering queue, and in our case, after the deferred shading pipeline and the post processing effects as well.

Every impostor shader implemented, has forward rendering shaders defined as well, that shade the geometry using the Phong shading model. Since we are only going to use the forward rendering shaders for transparent rendering, we also define the blending mode for forward rendering to be:

```
Blend SrcAlpha OneMinusSrcAlpha
```

and we also disable z writing. The alpha channel value is set as a constant number, but we could add it to the material property blocks. With this addition to the forward shaders, the only thing we have to do is to tell Unity that an impostor sphere or cylinder object is to be drawn as transparent. To do that, we are going to change the rendering queue of the object.

Rendering queues are defined by Unity in order to determine in which stage of the rendering pipeline each object belongs to [15], *Background*, *Geometry*, *AlphaTest*, *GeometryLast*, *Transparent*, *Overlay*. We are going to differentiate between opaque and transparent rendering, by using the *Geometry* and the *Transparent* rendering queue. As a result, each impostor object in the scene, has a function defined that can change it from opaque to transparent at runtime, by changing the rendering queue used.

User Interface

In Unity, there are essentially two modes of UI rendering, screen space and world space. Screen space UI is always rendered last, using the *Overlay* rendering queue. Since our main goal is to port the application in a Virtual Reality headset, screen space UI is not a choice. As a result, we are going to use an actual World Space User Interface inside the scene, and a World Space UI specifically mapped in the world, to appear as screen space UI.

World space user interface is rendered by default in the *Transparent* rendering queue, along with the transparent geometry. However, the order is important, since our transparent impostors, will also use the *Transparent* queue, and they will have z writing disabled. If we were to render the world space UI after the transparent geometry, then the UI would cover completely the transparent objects, even if the transparent objects were in front of the UI. Furthermore, we also want to be able to render a User Interface panel, that is on top of everything else, always visible to the user, or make the world space user interface visible to the user, no matter if it is currently hidden behind geometry objects. As a result, we have to impose an order within the *Transparent* queue.

Unity rendering queues are also defined by increasing numbers. *Background* is 1000, *Geometry* is 2000, *AlphaTest* is 2450, *Transparent* is 3000 and *Overlay* is 4000, where smaller numbers are rendered first. Furthermore, Unity allows to define rendering queues within these numbers, for example, objects with rendering queue 3000 will be drawn in the forward rendering pipeline, using their forward shaders, as well as objects with rendering queue 3001. However, objects with queue 3001 will be drawn after the ones with 3000. With this ordering in mind, we define the following rendering queues:

Transparent queue	Object	Z-Test	Z-Write
3000	World space UI	LEqual	Off
3001	Transparent geometry	LEqual	Off
3002	UI elements in the world (labels, lines)	Depends on the element	Depends on the element
3003	3D text labels in the world	Depends on the element	Depends on the element
3004	World space UI, made visible on top of geometry	Always	Off
3010	World-space Information panel mapped to screen space, always visible	Always	Off

The scene will have two user interfaces, the first one is an always visible information panel that is rendered on the top left of the camera view. This panel is the last thing rendered with the 3010 queue. The second user interface, is an actual world space UI, at a specific position in the world, that can be brought forward and rendered on top of the geometry, with the push of a button. When rendered normally in the world, this UI will use the 3000 queue, and when rendered on top of the geometry, it will use the 3004 queue.

Full rendering order

In order to compose the final scene, we only have to make sure that the post processing effects, like Ambient Occlusion and border highlighting are drawn before the forward rendering step, and our *Transparent* queue is drawn last. As a result, the full rendering order is the following:

1. Fill the GBuffer with the main scene geometry
2. Calculate Ambient Occlusion
3. Unity lightning calculations
4. Apply the border highlighting effect
5. Render the transparent queue

3.7 View frustum culling

View frustum culling is a CPU technique that culls geometry that falls outside the virtual camera's view frustum, since if rendered, it will not end up on the screen anyway. However, this view frustrum culling algorithm is not going to work properly for sphere impostors. As far as Unity is concerned, we have no spheres in the scene, but quads, and more specifically, quads that face a certain direction.

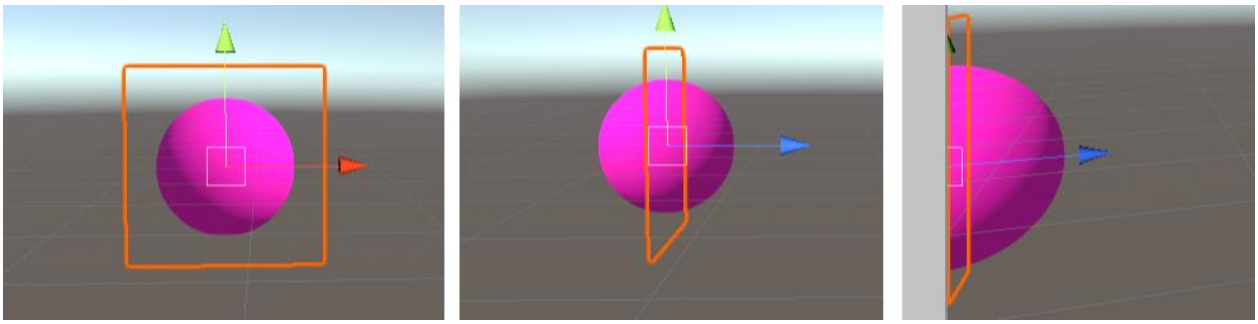


Figure 25: Spheres impostors, view frustrum culling problem

When looking at a quad down the *z* axis, view frustum calculation will be performed correctly, since the sphere is "contained" within the quad geometry. However, when looking at the sphere from the side of the quad, the orientation of the quad only changes in the vertex shader. View frustum culling will be performed with the original *z* facing quad. This produces an ugly effect of spheres being culled on the edges of the camera's frustum while still being visible. For example, in *Figure 25*, if the camera rotates just a bit to the right, the sphere will be culled.

This is not a problem for cylinder impostors, since the cube geometry tightly covers all the cylinder geometry generated in the fragment shader.

To avoid this effect on spheres, we will perform view frustum culling using a slightly higher field of view, and render the geometry with the default field of view. This will surely add some extra geometry being rendered, but the alternative of rotating all the quads in the CPU is strictly prohibitive.

To achieve the above effect, Unity allows us to attach a script to the virtual camera and declare the *OnPreCull()* function.

```
    void Start() {
        /* Grab components */
        cam = GetComponent<Camera>();
        cam_old_fov = cam.fieldOfView;
    }

    private void OnPreCull() {
        /* Change the field of view used for culling */
        cam.fieldOfView = cam_old_fov + fov_addition;
    }

    private void OnPreRender() {
        /* Set back old field of view for rendering */
        cam.fieldOfView = cam_old_fov;
    }
```

3.8 Unity Virtual Reality Rendering

Rendering for Virtual Reality devices, is supported in Unity with two modes, multi pass rendering, and single pass instanced stereo rendering. Multi pass rendering works by passing and drawing the scene twice, one for each eye, and it is supported by default for all Unity scenes and custom shaders. However, since the objects in the scene are drawn twice, the GPU and CPU times are expected to be at least doubled in multi pass XR rendering. The other technique is, single pass instanced stereo rendering which is a much faster VR rendering technique, since the scene is passed and drawn one time. This technique takes advantage of two features. Instancing, which allows the renderer to draw the same mesh multiple times, and stereo rendering, which allows the renderer to pass and cull the scene one time, and render it into a single packed texture for both eyes. However, not all Unity shaders automatically support this technique and since it's a more complex pipeline, not all graphics cards support it.

Single-Pass Instanced Stereo XR shader support

To add support to Unity custom shaders, we have to make sure that unlit shaders, post processing effects, and screen-space shaders correctly calculate vertex transformations to clip space, and correctly access the packed render texture for reading, since Unity is going to use a single double-width render texture for both eyes. Unity outlines the changes that must be made in the following articles [16], [17]. Since a VR ready computer capable of this technique was not available, the following paragraphs outline the steps that could be made to support it.

When performing stereo rendering, unlit (vertex/fragment) shaders must differentiate between the left and right eye, when transforming the vertices to clip space, since the left and right eye use different view matrices. Unity suggests that these vertex transformations to clip space are done through the standard *UnityWorldToClipPos(in float3 pos)* call. The only shader that is impacted from this, is the sphere impostor, since the sphere impostor shader calculates the camera oriented quad vertices in the vertex program in view space. However, if we examine the Unity built-in shaders, we can see that the *UnityWorldToClipPos()* function applies the standard transformation that our shaders do as well, by using the built-in View and Projection matrices, Appendix F: Stereo instancing Unity built-in matrix variables. This means that the sphere impostor vertex program that uses these matrices as well, does not need any change. If that is not

the case, we could abandon the calculation of the quad oriented vertices in the vertex program, and follow the same path like in the cylinder impostor, that is, render a geometry that tightly covers the perfect geometrical sphere, which is a cube for the sphere impostor. With this change, we can directly use the above function.

Post-processing effects must also change how they access the packed render texture. The only change that must be applied is when reading texel data, and use Unity's built-in function for transformation of *uv* coordinates *UnityStereoScreenSpaceUVAdjust()*.

To add support for stereo instancing, apart for the macros needed for regular instancing, we must also add to the vertex output data structure the *UNITY_VERTEX_OUTPUT_STEREO* macro. To know if a shader renders for the left or the right eye, Unity offers the variable *unity_StereoEyeIndex*, which has a different value for each eye. With this value, and with the necessary macros to set it up, the shader can differentiate between the two eyes.

3.9 Unity prefabs

The impostor sphere object, and the impostor cylinder object, can now be packed under a Unity prefab, which is an object ready to be instantiated in the world. Both the sphere and the cylinder impostor, expose an interface that allows the user to change their parameters (color, radius, highlighting, transparency etc.) by passing these changes through the material blocks, and thus at all times support instancing. However, the above prefabs have the following limitations:

First, shadow casting and shadow receiving is not implemented. Shadow casting requires the definition of a *ShadowCaster* pass, which has been explained briefly in the start of the Deferred rendering chapter. Shadow receiving requires some more careful handling, since the actual impostor fragments are generated in the fragment shader, and thus, we cannot simply use the built-in helper functions: *LIGHT_ATTENUATION* and *TRANSFER_VERTEX_TO_FRAGMENT* as they are. Impostors can support shadow receiving, if we carefully transform the world position of the fragment in the fragment shader, to the coordinate space defined by the light, by slightly tweaking the above macros.

Another limitation of these prefabs is that the forward rendering pass is not compatible with factory post processing effects.

3.10 Performance and Unity evaluation

In the first part of this section, the performance benefits from using the impostor techniques will be explored, while in the second section are demonstrated the advantages and disadvantages of using Unity.

Performance

In order to evaluate the performance benefit of using impostors to render spheres and cylinders, we spawn the same scene, one time by using Unity spheres and cylinders, and another time using impostors, for the following models:

- 1TES 1435 atoms, 1092 bonds
- 4F0H: 5030 atoms, 3895 bonds
- 1S5L: 45945 atoms, 42644 bonds

All benchmarks were run in a low-end laptop computer with the following specs: i5-8250U, NVIDIA MX150 2GB, in 1920x1080 resolution, since high performance Desktop computers were not available. The results are the following:

1TES	Triangles rendered	CPU main (ms)	Render thread (ms)	FPS
No impostors, No Instancing	1.2M	5	2	200.1
No impostors, Instancing	**1.2M**	**4.3**	**0.8**	**230.2**
Impostors, No instancing	18.5K	6.1	3.7	161.5
Impostors, Instancing	**18.5K**	**4.8**	**1.1**	**210.4**

4FOH	Triangles rendered	CPU main (ms)	Render thread (ms)	FPS
No impostors, No Instancing	4.2M	14.7	5.6	68
No impostors, Instancing	**4.2M**	**12.9**	**2**	**77.4**
Impostors, No instancing	63.0K	18.3	10.2	54.7
Impostors, Instancing	**63.0K**	**13.9**	**1.7**	**72.1**

1S5L	Triangles rendered	CPU main (ms)	Render thread (ms)	FPS
No impostors, No Instancing	38.7M	332.8	222.4	3.0
No impostors, Instancing	**38.7M**	**126.8**	**14.5**	**7.9**
Impostors, No instancing	605.3K	225.9	118.3	4.4
Impostors, Instancing	**604.5K**	**131.6**	**12.1**	**7.6**

The key outtakes from the above numbers is that, first, if the scene has thousands of objects for rendering, instancing, or any other technique for render call batching is a must. Second, impostors will only yield a performance benefit when the GPU is not able to render the geometry. For example, in the small model, the GPU is able to render 1.2M triangles, and thus, impostors don't improve the performance. However, as the models get larger, the impostors scale better in rendering times. Of course, by using impostors, we also render spherical geometry with much higher fidelity.

The main problem however, in all executions, is the CPU main time. In Unity, having thousands of objects in the scene, even if they are marked as static, even if they have no *Update()* function attached, will worsen performance. For example, the following image is a snapshot of the Unity profiler, for a specific frame when rendering the *1S5L* model, with impostors and instancing:

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	92.8%	0.0%	2	0 B	129.80	0.11
Camera.Render	91.2%	0.0%	1	0 B	127.60	0.12
Drawing	57.5%	0.0%	1	0 B	80.40	0.07
Render.OpaqueGeometry	50.9%	0.7%	1	0 B	71.23	1.07
RenderDeferred.GBuffer	49.9%	0.0%	1	0 B	69.88	0.07
WaitForJobGroupID	29.3%	2.4%	1	0 B	41.08	3.44
RenderDeferred.GBuffer	20.5%	17.7%	1	0 B	28.71	24.87
Clear	0.0%	0.0%	1	0 B	0.00	0.00
RenderTexture.SetActive	0.0%	0.0%	3	0 B	0.00	0.00
RenderDeferred.Reflections	0.0%	0.0%	1	0 B	0.08	0.08
RenderDeferred.Lighting	0.0%	0.0%	1	0 B	0.07	0.01
RenderDeferred.ReflectionsToEmissive	0.0%	0.0%	1	0 B	0.07	0.02
RenderDeferred.CopyDepth	0.0%	0.0%	1	0 B	0.04	0.04
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00
JobAlloc.Grow	0.0%	0.0%	1	0 B	0.00	0.00
Render.Prepare	6.4%	6.4%	1	0 B	9.01	9.00
Camera.RenderSkybox	0.0%	0.0%	1	0 B	0.03	0.02
Render.TransparentGeometry	0.0%	0.0%	1	0 B	0.03	0.01
Camera.FireOnPostRender()	0.0%	0.0%	1	0 B	0.01	0.01
RenderLoop.CleanupNodeQueue	0.0%	0.0%	2	0 B	0.00	0.00
Camera.ImageEffects	0.0%	0.0%	1	0 B	0.00	0.00
Render.MotionVectors	0.0%	0.0%	1	0 B	0.00	0.00
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00
CullResults.CreateSharedRendererScene	20.1%	4.8%	1	0 B	28.24	6.71
DestroyCullResults	9.5%	0.2%	1	0 B	13.40	0.38
Culling	3.8%	0.0%	1	0 B	5.35	0.01

Figure 26: Unity profiler for 1S5L

In the above image, we can see how much time rendering and culling require. For that specific frame, CPU main time was 153.3 *ms*, while the GPU time was 14.6 *ms*, which means that the bottleneck in performance is managing thousands of objects in the scene. In order to increase performance, we would have to employ techniques that aim to lower the number of objects in the scene, for example, partitioning the world.

Furthermore, given that the Unity GameObject is a relatively heavy structure, we could try to use a more lightweight game engine, or even create a custom one, that does the bare minimum required.

Unity advantages and disadvantages

Using Unity comes with a long list of advantages, where the most important ones are debugging superiority and development speed. When writing the shaders implemented, and managing the order of rendering, one of the most valuable features was the Frame Debugger. In the frame debugger, one can carefully examine the order in which render calls are issued, the parameters that the shaders have, and the results of the rendering in a single place.

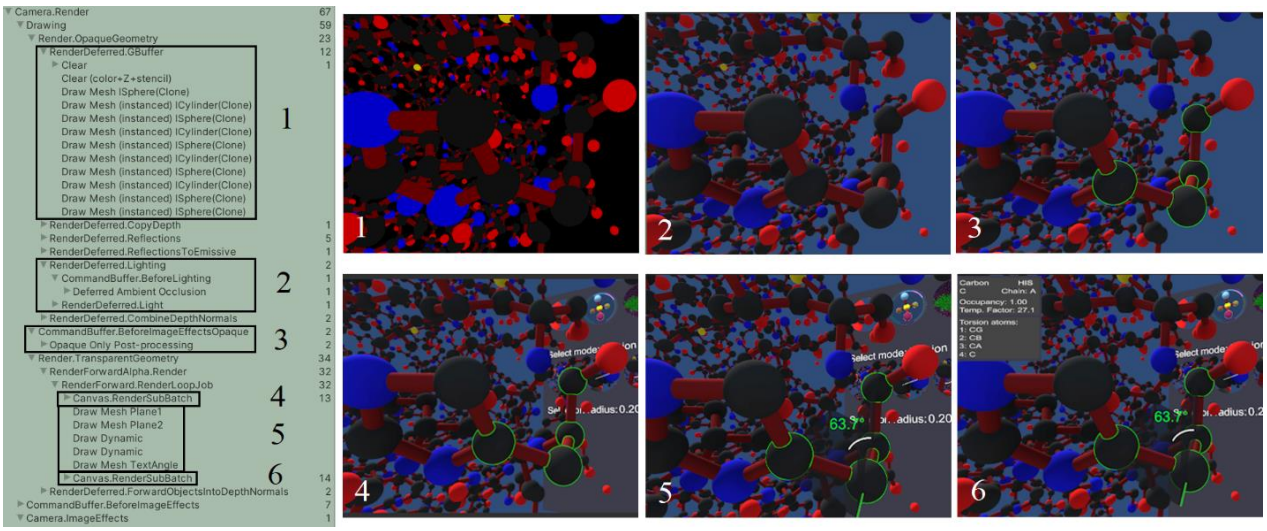


Figure 27: Debugging rendering order using the Frame Debugger

When it comes to development speed, one can jump right into the implementation of his work, while Unity takes care of Input, Output, disk IO, VR peripherals, and also takes care of several other “invisible” features. For example, if we were to implement our own custom engine, we would definitely have to implement several acceleration data structures, to ease culling, scene traversal and ray casting. Furthermore, we would also have to implement a complete rendering pipeline that involves both deferred, post processing effects, and UI forward rendering.

The disadvantages of Unity in this project, can be summed up to the fact that there are several restrictions when using the engine. For example, Unity is very centralized around the GameObject structure. For example, it is not possible to have colliders to perform ray casting, without having GameObjects, and it is not straightforward to render the same GameObject multiple times with different shaders. Another disadvantage of writing Unity shaders is that, Unlit shaders, that is Vertex/Fragment shaders, don’t have extensive documentation, and many times one must rely on diving into the code of the Unity built-in shaders.

4. Interaction

Interacting with a Virtual Reality scene, usually involves one or more 3D selection techniques, with the most common being, especially in Head Mounted Displays, ray casting. However, ray casting a specific object with a hand-held controller, can sometimes be very challenging, especially in complex and cluttered areas. As a result, this second part of the work, will be devoted on a novel navigation technique within a molecular scene, that does not need ray casting. This navigation technique will instead use simple 2D direction input, that can be given with the arrows in a keyboard, or with a touchpad in a controller, that will allow the user to select atoms without ray casting to them. We are going to then use this navigation technique to visualize properties of the molecular scene, like the distances between atoms, the angles between bonds, and the torsion angles formed by multiple atoms.

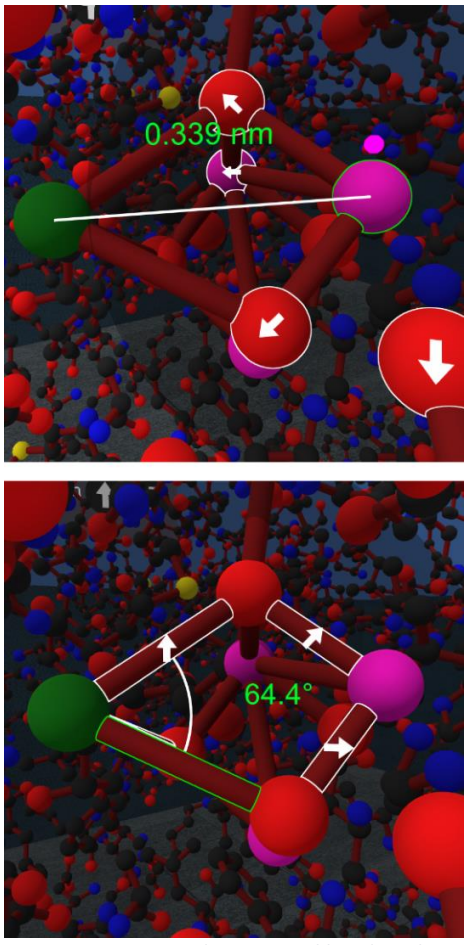


Figure 28: Atom distance and bond angle visualization

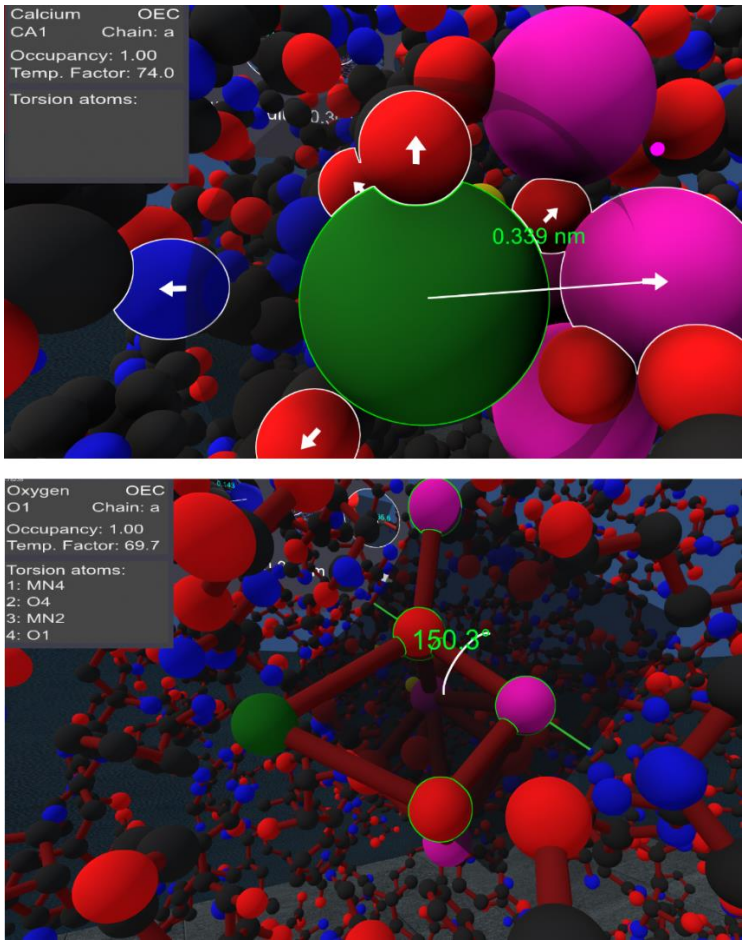


Figure 29: Atom distance in space filling model, and torsion angle visualization

4.1 3D selection

To select an object in a VR environment, we have to employ a 3D selection technique. F. Argelaguet and C. Andujar in [18] provide a thorough list of 3D selection techniques, along with their advantages and disadvantages. In this work, we mainly focus away from 3D selection, and thus, we will build upon a simple 3D ray casting technique.

To be able to ray cast into the scene and get back the closest intersection, either if it is a sphere, or a bond, or a UI element, we are going to attach Unity colliders onto the impostor objects. On the sphere impostor we attach a Sphere Collider, in the Cylindrical impostor we attach a capsule collider and in the UI elements we attach Box Colliders. Capsule colliders in bonds, have a slightly bigger size to ease ray casting.

4.2 Information panel

The information panel, is a world space canvas, specifically mapped to always be on the top left of the camera view, that displays information about the selected atom.

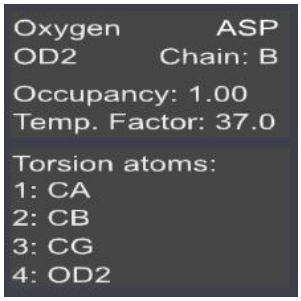


Figure 30: Information panel

The information panel displays the type of the current atom selected, and the residue that it belongs to in the first row, in the second row it displays the exact atom name within the residue given the molecule's and atom's nomenclature, as well as the secondary structure that this atom belongs to. The third and fourth row display the occupancy and temperature factor of the atom, as parsed from the *.pdb* file. The torsion atoms section is used to mark atoms when visualizing torsion angles. This object is solely for the purpose of information visualization, and the user will not have to interact with it in any shape or form.

The information panel is always rendered on top of everything else in the scene and it's the last thing rendered in the transparent queue, with the Z-Test set to *AlwaysPass*.

To map the position of the world space panel on the top left of the camera, we make the panel a child of the camera object, and calculate the position of the label in the local coordinate system defined by the camera, with the following algorithm:

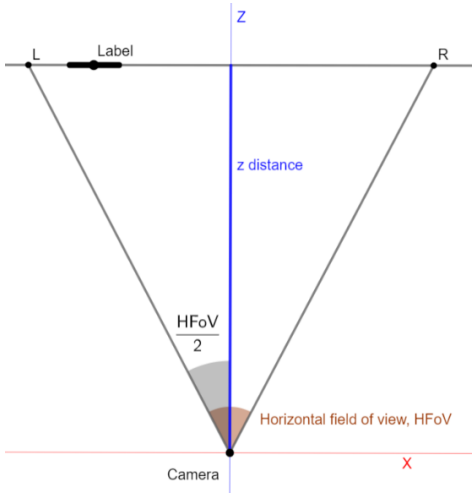


Figure 31: Calculating horizontal position of the information panel

$$LR = (\text{z distance}) * \tan\left(\frac{HFoV}{2}\right), \quad Label.x = -\frac{LR}{2} + horizontal\ offset + \frac{panel\ world\ width}{2}$$

With a similar equation with the above, and using the vertical field of view, the height of the panel and a vertical offset, we can calculate the *y* coordinate of the position of the label. The *z* coordinate is fixed at *z distance*. After setting the position, the panel is rotated to face the camera and since the object is a child of the camera, the initial position and orientation is preserved along the movement of the camera.

With these calculations, we can precisely control the position of panel in the camera view, and "zoom in" or "zoom out" the panel by changing the *z distance* parameter. Furthermore, since the calculation is automatic, porting the application in different resolutions is much easier, since we only have to adjust the *z distance* parameter, because going in lower resolution will "zoom in" the label, and higher resolutions will have the opposite effect.

4.3 Visualizing residues and chains

With the use of ray casting, the user can visualize residues and chains within the molecular model. To visualize residues, we highlight all the atoms that belong to the residue of the atom currently ray-casted by the user. While this technique works well to pinpoint small structures within a big group of molecules, it is not very effective to visualize bigger structures like chains. Towards that, to visualize chains, we change the color of all the atoms that belong to the chain of the currently ray casted atom, and at the same, highlight the atoms that belong to the same residue. In both cases, the information of the ray casted atom is displayed on the information panel.

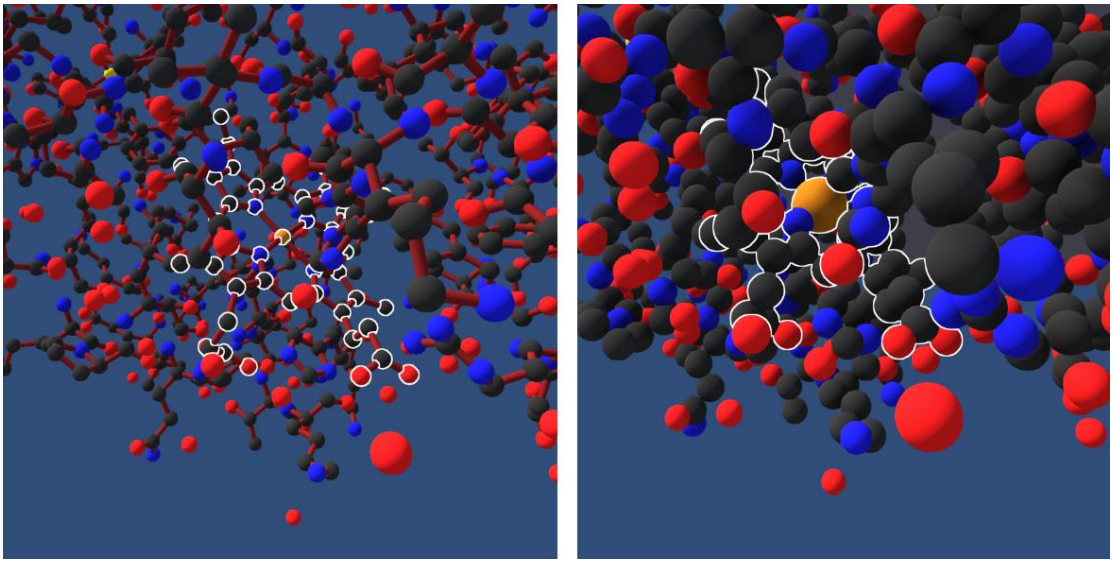


Figure 32: Residue visualization in ball-and-stick model (left), and space-filling model (right)

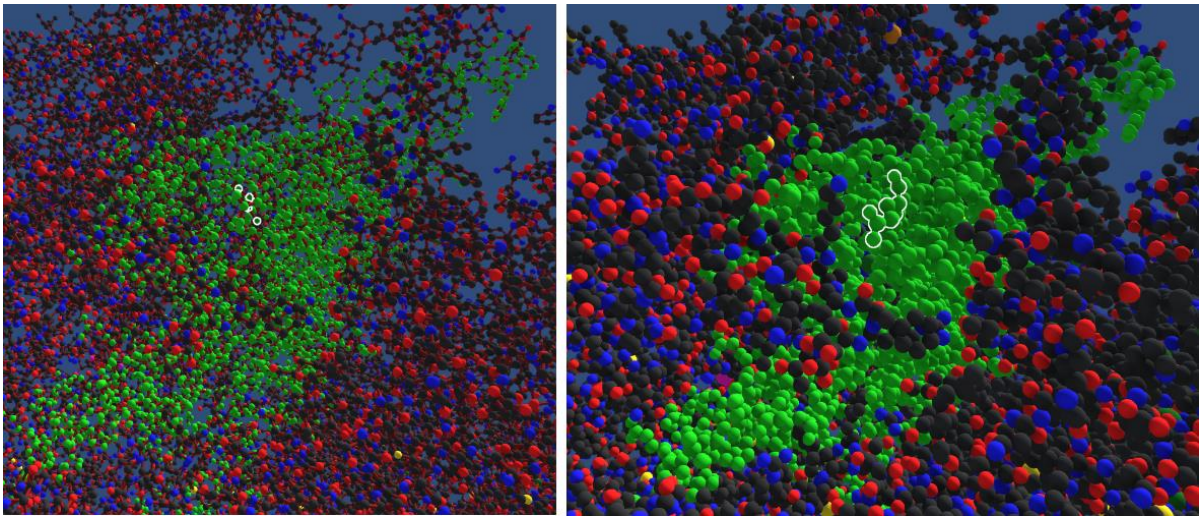


Figure 33: Atom chain visualization in ball-and-stick model (left) and space-filling model (right)

4.4 2D atom selection

Selecting atoms with ray casting in a 3D VR environment, can be challenging in a very cluttered scene with many objects. As a result, we implemented a navigation method between close atoms that utilizes 2D direction input that can be given by arrow keys, in a keyboard configuration, or joystick control in a VR environment.

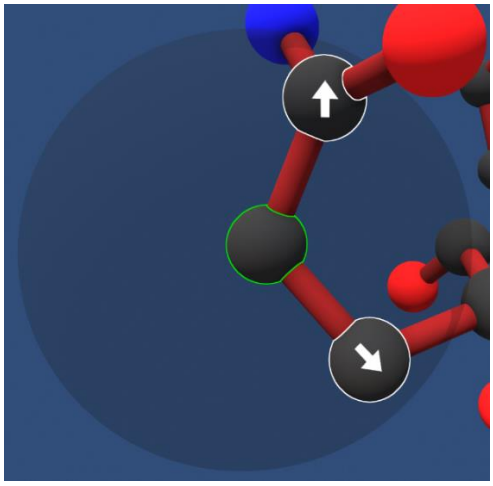


Figure 34: 2D navigation within close atoms

The idea that, the user can select a specific atom with a 3D selection technique with the push of a button, and then jump to the atoms within a region by using the common eight directions in 2D, up, down, left, right, and their combinations. For example, in *Figure 34*, once the user has selected the atom highlighted green, he can navigate to the atom on the top by giving an upwards direction and triggering a *select button*, or navigate to the bottom right atom by giving a bottom right direction. The user is also able to change the radius of the selection sphere in real-time, so that he can potentially reach atoms that would otherwise be inaccessible.

In this navigation scheme, it is important to give the following visual feedback. First, the region of atoms that can be potentially reached, which is given by the transparent sphere that shows the radius of the navigation. Second, the atoms that can be in fact navigated to, using the 2D direction input at the current frame. These atoms are highlighted with white border. The above, are two different groups of atoms. The first only depends on the selected atom and the radius of the selection sphere, while the second also depends on the camera angle. It is possible for a given camera angle, that many atoms within a spherical region to be mapped to the right direction for example, but only one of them will be chosen to be available for navigation.

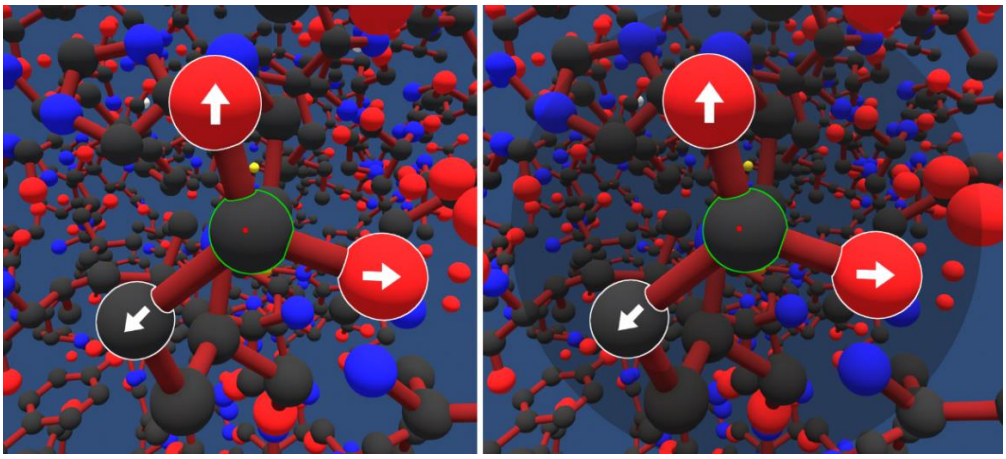


Figure 35: Atom selection with and without rendering a sphere to indicate the region of selection

The transparent sphere can also isolate the selection of atoms, when the background has a lot of "noise". It is rendered with a transparent impostor sphere using the forward rendering queue and keeping the farthest intersection point when performing ray sphere intersection, to simulate front face culling.

Third visual feedback that must be given, is which atom is the currently selected atom. This sometimes is obvious, but in more cluttered scenes, it is important to distinguish it from the rest, and thus, highlight it with a different color border. Also, the selected atom, is the atom whose information is currently displayed in the information panel.

Last but not least, an important visual cue is an indication about the direction that must be chosen in order to navigate to an atom, for example bottom left direction. Towards that, we implement two methods to provide such feedback. The first method is an arrow indication, and the second method is color coding.

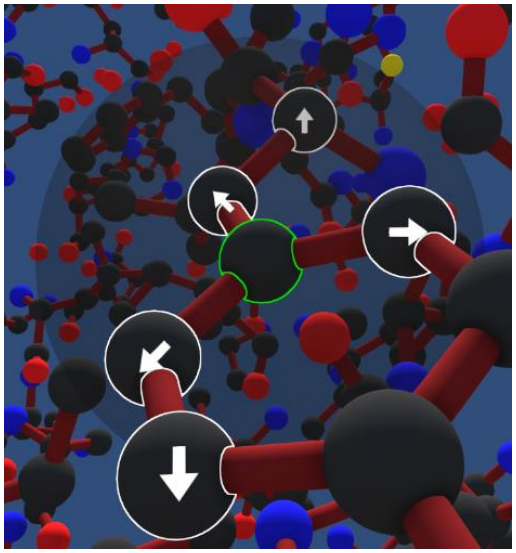


Figure 36: Atom navigation using arrows in ball-and-stick model

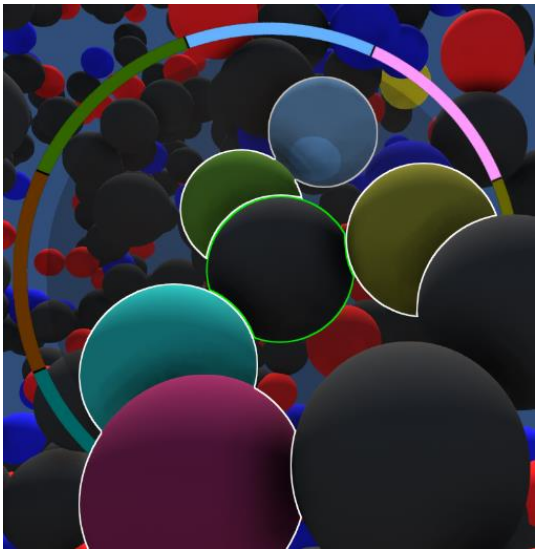


Figure 37: Atom navigation using color coding in space filling model

Mapping to 2D

The basis of these two selection techniques is the mapping of the atoms onto a 2D plane, so that we can use the mapping to get the 2D direction that each atom corresponds to. To achieve that, we declare a new coordinate system. Let C be the camera position in world space coordinates, and A be the world space position of the selected atom. The new coordinate system will then be:

$$New\ origin = A, \qquad New\ \vec{Z} = A - C, \qquad New\ \vec{Y} = Camera\ up, \qquad New\ \vec{X} = \vec{Z} \times \vec{Y}$$

We will transform all the atoms within a spherical region into the above coordinate system, and we will use the new (x, y, z) components of each sphere, to calculate the 2D direction.

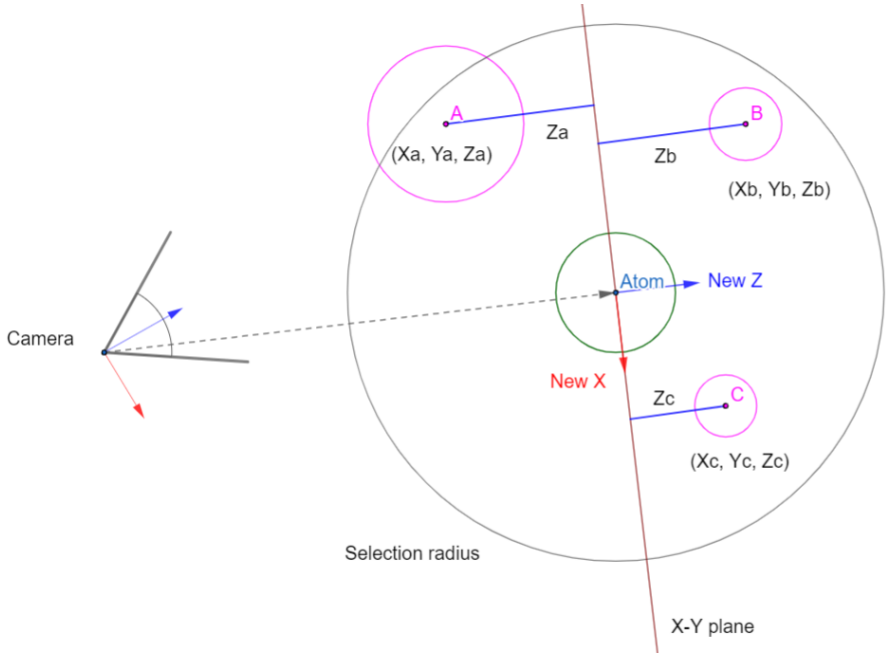


Figure 38: Top down look of the new coordinate system

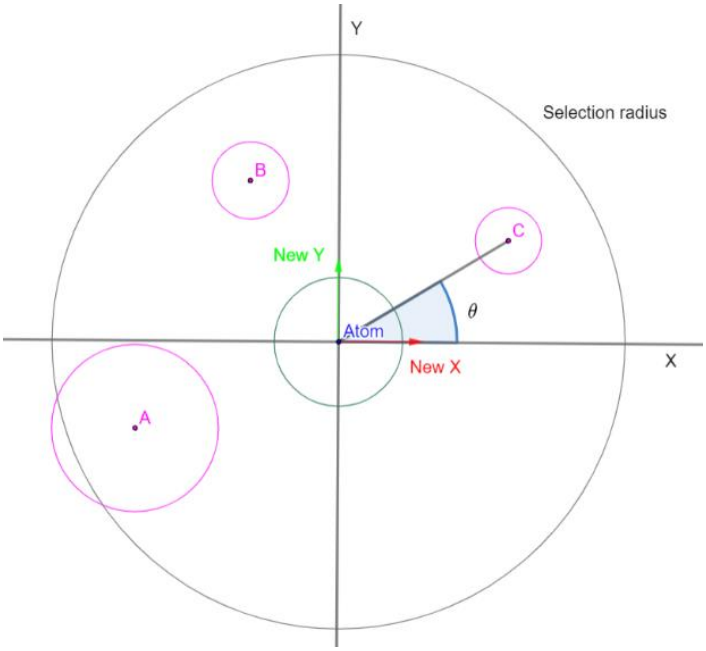


Figure 39: View of the new coordinate system from the perspective of the camera

To calculate in which direction each atom corresponds, we will use the angle defined by the x and y components of its coordinates in the new coordinate system:

$$\theta = \arctan\left(\frac{y}{x}\right)$$

By discretizing the circle into eight octants, we can use the above angle to calculate in which octant – direction each atom corresponds to.

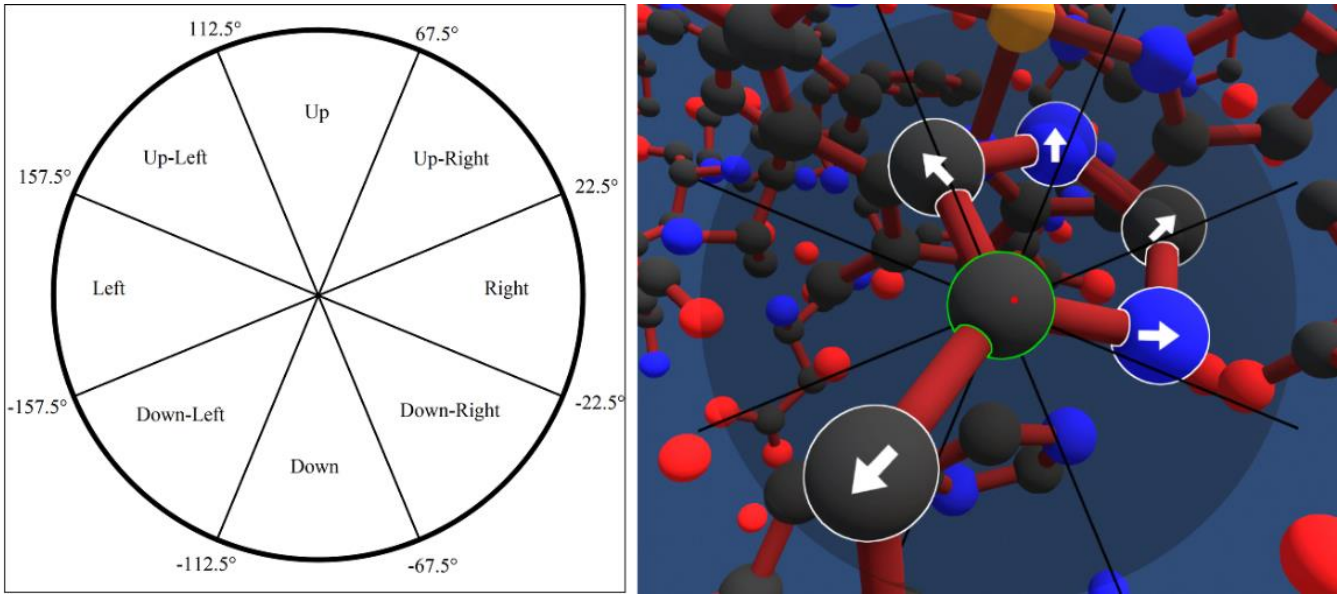


Figure 40: 2D direction based on the XY angle of the atom in the new coordinate system

This 2D mapping is calculated at every frame, and thus the direction per sphere changes as the camera position changes. If multiple atoms fall into the same octant, we are going to keep the atom that is closest to the new xy plane. This distance is given by the absolute value of the z component, of the coordinates of the atom in the new coordinate system. Since the orientation and the position of the xy plane depend on the camera angle and position, the atom that is selected using this metric can easily change when the viewer changes its position and orientation. This makes it easy for a combination to exist, of a viewing angle and camera position, so that every atom within the selection sphere can be navigated to. If we were to use another metric, for example, the Euclidean distance of the atom to the center, then the atoms that can be navigated to would be the same, most of the time, since the metric would only rely on the positions of the atoms, which are constant.

Arrow direction indicators

A straightforward way to indicate which direction must be chosen in order to navigate to a specific atom, is to draw the direction in the front of the atom. The arrow object, is a quad whose position is always in front of the atom, by using the vector from the atom towards the camera and at distance equal to 1.2 times the radius, based on the visualization method used. The quad is rotated per frame to always face the camera.

The disadvantage of this technique is that, it is possible for very complex scenes, that the arrow indication is hidden.

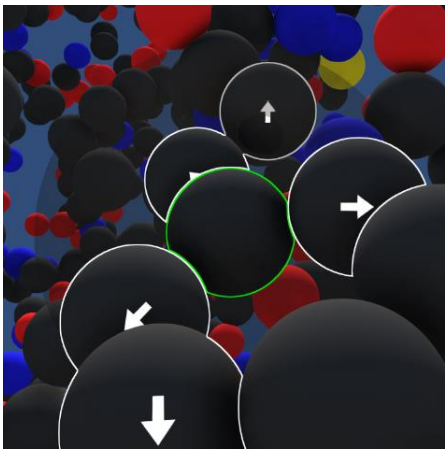


Figure 41: Arrow indication in the space filling model

This problem is more prevalent in the space filling model, where atoms are much larger and can easily hide the arrow indication.

Color coding directions

Another technique that we can use to visualize the direction for each atom, is color coding. We already saw an example of this in *Figure 37*. The disadvantage of this technique is that it is sometimes confusing having many different colors simultaneously on the scene, and that we have to carefully pick the colors so that they don't interfere with the CPK coloring scheme. The color-coding visualization can only be useful in complex scenes, where the arrow indications might be hidden. In the ball-and-stick model, the arrow indication is always both simpler and more straightforward.

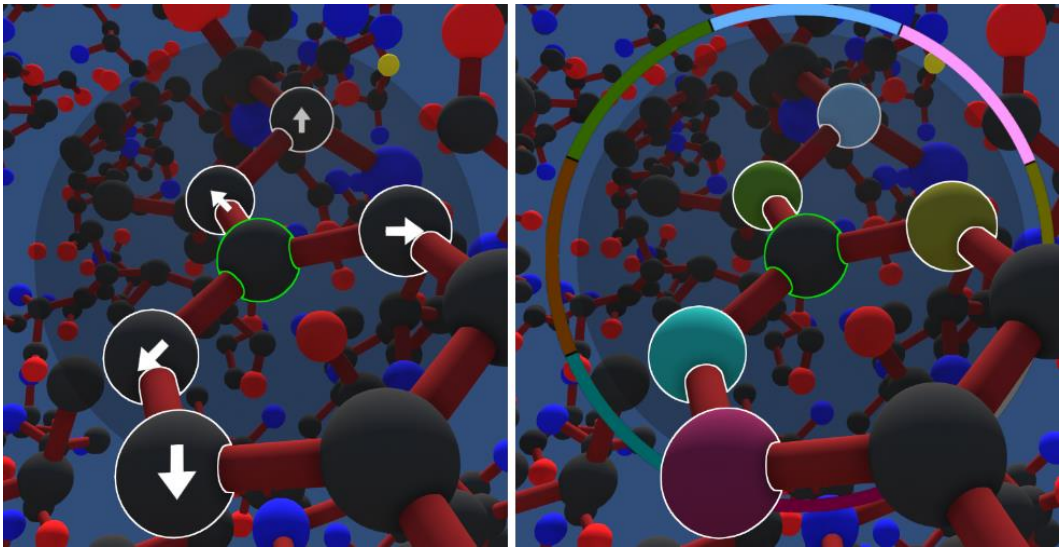


Figure 42: Comparison between the color circle and arrows in the ball-and-stick model

However, we believe that once the user is accustomed to using the keyboard or joystick control, the direction selection becomes easier, and none of the two indications might be needed.

4.5 Atom distances

The user can interact with the atoms in the scene, and by using the above selection technique, display the distance between any two atoms.

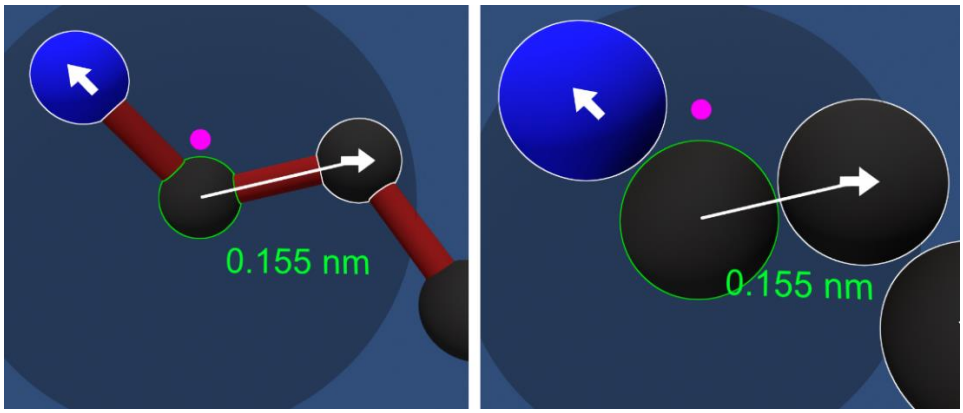


Figure 43: Atom distance visualization, ball-and-stick(left), space filling with covalent radius (right)

Since the atom distance makes sense not only in consecutive atoms, but in any combination of atoms, the interaction must allow the user to select any two atoms in the scene. Thus, it is important to introduce a new *mark button*, apart from the 2D direction input control and the *select button*. The *mark button* will mark an atom in the scene, and the last two marked atoms will be used to calculate the atom distance. A marked atom in the scene, is visualized with a pink dot on the top. The mark dot is rendered with depth testing disabled, and thus, it is not possible to be hidden in the space filling visualization mode.

To render the atom distance, we will render a line connecting the two atoms, as well as a label with the distance number. To render the line, we will use a custom unlit shader that ignores depth testing, since if there is a bond connecting two atoms, the line would not be visible. The world space label with the

distance number, is a quad always facing the camera, whose position is calculated with the cross product between the vector that connects the camera and the middle of the distance, and the vector that connects the two atoms. The position of the label is static in the world, i.e. does not change depending on the camera position, but rather calculated once. In contrast, the position of the direction arrows in the 2D navigation, are constantly calculated to be in front of the atom. Furthermore, since the position of the number label is not guaranteed to be visible and not occluded, the rendering ignores depth.

4.6 Bond angles

The user can select consecutive bonds and display the angle between them. The angle between bonds is an important piece of information, as it is part of the molecule geometry that influences many properties of a substance, for example, reactivity, polarity, phase of matter, color, magnetism and biological activity. The bonds that the user can select and display the angle between, must have the same origin, so that the angle value is meaningful.

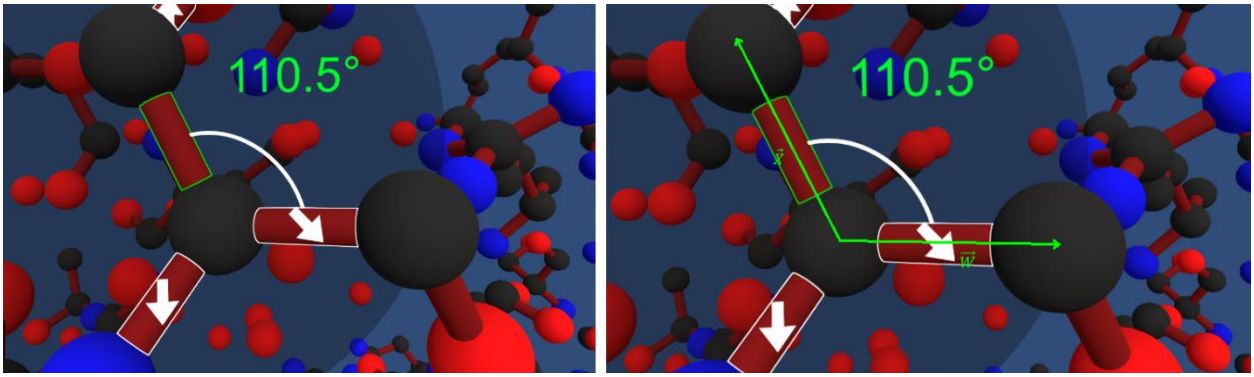


Figure 44: Visualization of angle between bonds(left), vector calculation(right)

To select consecutive bonds, we can adapt the 2D selection technique used previously for navigation. Furthermore, since the bond angle only makes sense between bonds with a common origin, we don't need to have the additional *mark button*, but we can have the bond angle displayed continuously between the last two bonds visited during 2D navigation.

The molecular angle is always displayed in the [0 – 180) range, as a result, we have to always make sure to display the angle in that range and draw an arc that always connects the smallest angle of two (instead of its complementary), no matter in which order the bonds are selected. To do that, we calculate the direction vectors of the bonds, $\vec{X}$ and $\overrightarrow{W}$ in *Figure 44*, always with the regard to the same origin, which is their common atom, disregarding the orientation of the 3D objects that encapsulate the two bonds.

That way, the angle can be calculated as:

$$angle = \ arccos(\frac{\vec{X} * \overrightarrow{W}}{|\vec{X}| * |\overrightarrow{W}|})$$

With the direction vectors $\vec{X}$ and $\overrightarrow{W}$, and the above angle value, we can apply the following formula to draw the arc:

$$\vec{Z} = \vec{X} \times \overrightarrow{W}, \quad \vec{Y} = \vec{Z} \times \vec{X}$$

$$P(\theta) = R * \cos(\theta) * \vec{X} + R * \sin(\theta) * \vec{Y}$$

where $\theta \in [0,\ angle]$, and R is the desired radius of the arc. The above formula gives us points along the arc in the local coordinate space of the arc object, which is placed at the point of intersection of the two direction vectors $\vec{X}$ and $\vec{W}$.

To draw the arc, we use a resolution of 30 points along the arc connected with a line, using Unity's *LineRenderer* tool. To have the arc rotated and moved in accordance to the whole model, we have to make sure to always transform the local coordinate space arc points to world space coordinates, every time there is a translation or rotation of the parent object, since Unity's *LineRenderer* draws lines whose coordinates are in world space.

The position of the angle number label, is calculated by using the vector that connects the arc origin with the middle of the arc, i.e. $P(\frac{\theta}{2})$, and expanding along that direction using the required radius of the arc, and multiplication factor of 1.8. The rendering of the label is done by a custom shader that takes into consideration Z-Testing, since Unity's default 3D text rendering shader does not.

4.7 Torsion angles

A torsion angle, is the angle defined as a particular example of a dihedral angle, describing the geometric relation of two parts of a molecule joined by a chemical bond [19]. Every set of three non colinear atoms in space define a plane. Thus, a group of four consecutively-bonded atoms, A-B-C-D, define two planes, A-B-C and B-C-D. The torsion angle then, is the dihedral angle between these two planes.

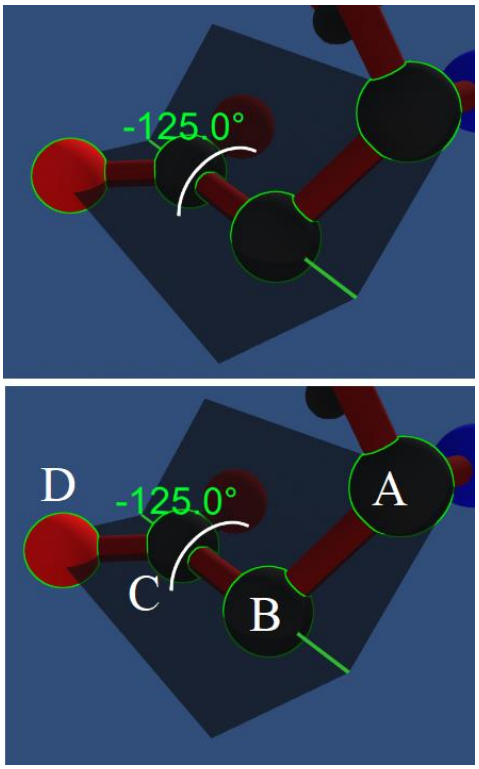


Figure 45: Torsion angle visualization (up), selected atoms order (down)

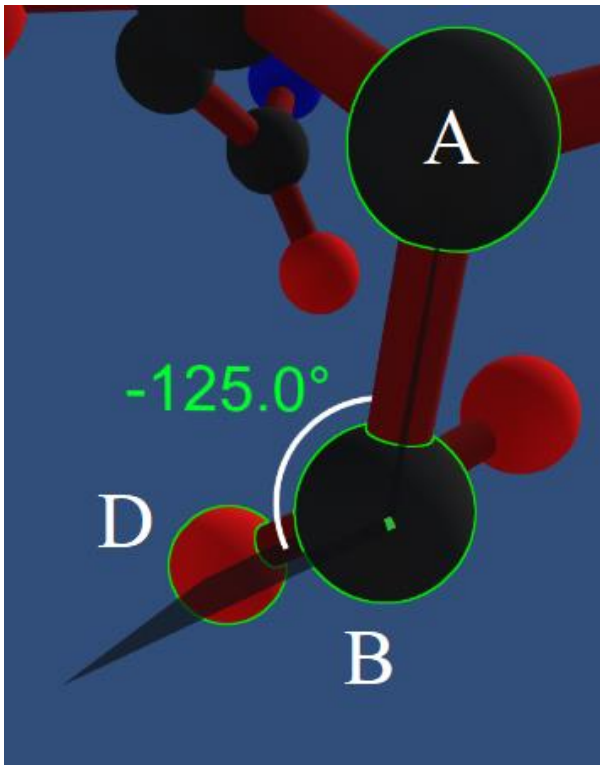


Figure 46: Torsion angle visualization, aligned view

In its absolute value, the angle is between 0° and 180°, however, the torsion angle between groups A and D is considered to be positive if the bond A-B is rotated in a clockwise direction through less than 180° in order to eclipse the bond C-D. A negative torsion angle requires rotation in the opposite, counter-clockwise sense. The symmetric applies for the angle between D and A.

The torsion angle is a very important piece of information in stereochemistry, and it is used widely in molecular conformation [20], as a result, we want to be able to calculate it and visualize it effectively.

Selecting atoms

To select the atoms that will form the torsion angle, we can use the same selection technique that we used in the previous chapters. Just like in the bond distance visualization, the user can navigate through the atoms in the scene freely, and use the *mark button* to mark atoms for the torsion angle. Since the order here matters, we are going to display the marked atoms in the information panel, in the order in which they are marked. Once the user has selected four atoms, the torsion angle will be calculated and displayed automatically.

Calculating the dihedral angle

The order in which the atoms are selected is important, since in an A, B, C, D configuration, the B, C atoms are the ones that participate in the formation of both planes, and also define the axis of rotation for the torsion angle. As a result, if the user has selected the atoms in the above order, the calculation of the torsion angle is done with the following formulas:

$$\overrightarrow{n_1} = PlaneNormal(A, B, C), \qquad \overrightarrow{n_2} = PlaneNormal(B, C, D)$$

$$angle = \arccos(\overrightarrow{n_1} * \overrightarrow{n_2}), \qquad sign = sign(\overrightarrow{n_2} * (A - B))$$

Visualizing the dihedral angle

To visualize the dihedral angle, we are going to render transparent planes that represent the planes defined by the atoms, we are going to highlight the atoms that form the dihedral angle, and we are also going to draw an arc, with the same algorithm like the bond angle.

To render the transparent planes, we are going to build the meshes of the planes from scratch, i.e. we will calculate the world space coordinates of the plane vertices, using the world space coordinates of the atoms. This is going to give us more control to the position, scale and orientation of the meshes, compared to using Unity's plane mesh object, and trying to position and orientate each plane according to the atoms. Thus, we calculate the world space coordinates of the points PA, PB, PC, PD, in Figure 47, and construct two back and two front facing triangles for a single plane.

This algorithm is run symmetrically for the atoms *C-B-A* and *B-C-D*, so that *Atom1* and *Atom2* always refer to the axis atoms.

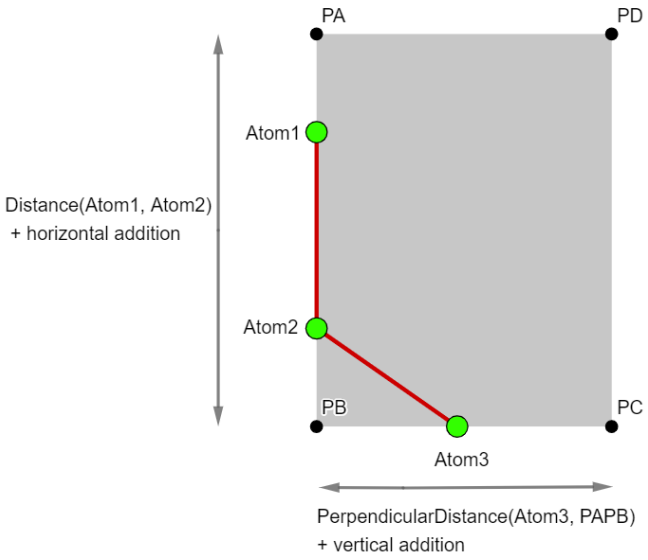


Figure 47: Torsion plane construction

To draw the arc, we use the same arc object used to visualize the bond angle, by defining the following $\vec{X}$ and $\vec{W}$ directions, originating from the middle of the atoms that form the common axis:

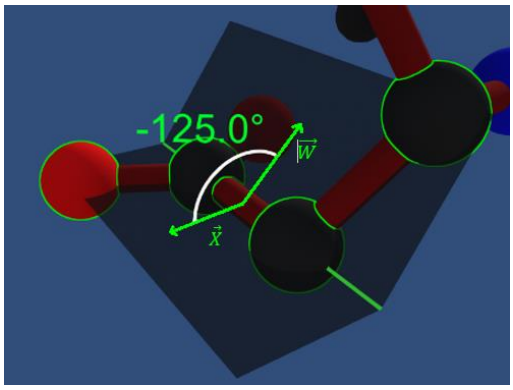


Figure 48: Torsion angle arc rendering

4.8 Translating and rotating

Translating and rotating the whole model is supported for all of the above interaction methods.

Since rendering the geometry requires no additional transformation, other than the Model matrix computed by Unity, we need not re-calculate anything in order for the impostor objects to be rotated or translated in real-time. We simply group all atoms and bonds under a single parent object called *Atoms*.

The 2D navigation technique, re-maps the atoms within the spherical region to the new coordinate system in each frame by default, since the 2D direction changes based on the camera position and orientation. As a result, the 2D directions also respond to changes in rotation and position of the object itself.

Text labels, arrow direction labels, color circles and torsion planes, are Unity objects grouped under the same parent, and as a result, their rotation is handled by Unity.

The only thing that we must make sure is rotated and translated correctly along with the *Atoms* object, are lines drawn through Unity's *LineRenderer* object, since this object requires that the coordinates are given in world space. Thus, every time a line is drawn, we calculate the points of the line in the local object space, and at real-time, check if there has been a change in the Model matrix of that object, and if it has, re-transform the local object space points to world space, and pass them to the *LineRenderer*.

4.9 World space widget

In order to facilitate interaction, the scene will also have another object to allow the user to change the parameters of the visualization. This panel, is an actual world space panel, i.e. has a specific position in the world unchanged, and has world space virtual buttons with which the user can interact with, and change modes of interaction.

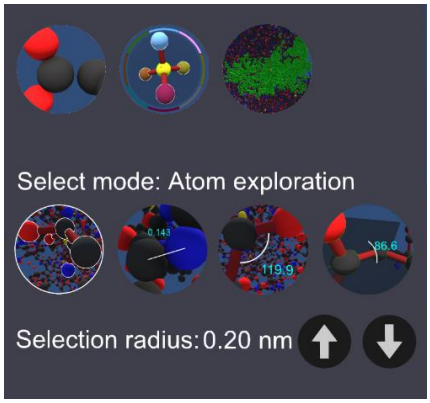
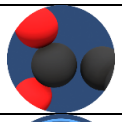
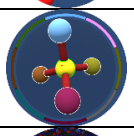
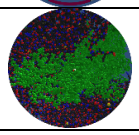


Figure 49: World space interaction panel

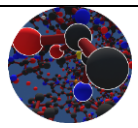
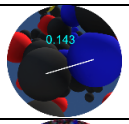
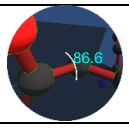
This widget, has the ability to be drawn in front of everything else in the scene, in order to facilitate interaction in cluttered environments. The ability to bring forth this widget, can me mapped into a physical button. When that button is pushed, this panel is drawn on top of everything else in the scene, except for the information panel, and the user can perform ray casting only against that panel. That way, the user can change parameters and modes of interaction, even when the panel is occluded.

Modes of interaction and visualization parameters

There are three configurable options:

	This option is used to change the visualization model of the atoms. The user can interact with it with ray casting, and in real time, change the visualization between space filling and ball-and-stick. The default is ball-and-stick model.
	This option is used to change the method used to display the direction that must be chosen, when navigating between atoms or bonds. The default is arrow direction labels.
	This button is used to change the exploration method between visualizing chains, and visualizing residues. The default is visualizing residues.

and four modes of interaction:

	**General atom exploration**: In this mode, the user can ray cast towards the atoms of the model, in any combination of model visualization and chain or residues visualization, and when an atom is selected, navigate through the neighbors while visualizing the information of the currently selected atom.
	**Atom distances**: In this mode, the user can ray cast towards the atoms of the model, in any combination of options, and when an atom is selected, navigate and mark atoms for distances, as explained in the atom distances section.
	**Bond angles**: In this mode, the user can ray cast towards the bonds of the model, and when a bond is selected, navigate within the bonds while displaying the angle between them, as explained in the bond angles section. This mode is not compatible with the space filling visualization, or the chain or residue visualization.
	**Torsion angle**: In this mode, the user can ray cast towards the atoms of the model, in any combination of options, and when an atom is selected, navigate and mark atoms for torsion angle freely, as explained in the torsion angle section.

In general, the user can configure any of the above parameters and modes, in any combination. For example, the user can measure atom distances either in the space filling or the ball-and-stick model. Navigate within bonds and atoms by either using the arrow directions or the color circle. The visualization of chains or residues refers to the first ray casting call in any mode, before selecting an atom. For example, when in the atom distances mode, and while ray casting towards the model before selecting an atom, then the user can either visualize residues or chains. Of course, there are some incompatible options, for example, the bond angles mode does not make any sense along with the space-filling mode, since there are no bonds. Nonetheless, the user is free to interact with the model in that way.

Furthermore, in order to minimize the ray casting operations, when the user switches between modes, the previously selected atom will remain. For example, switching from atom distances to torsion angles, will keep the currently selected atom, and the user can start visualizing torsion angles without a selecting a new atom, by just navigating to the neighbors with 2D controls. When the change of mode involves change between selected atom and selected bond, then the nearest object in either case is fetched, if such exists.

The final option in this panel gives the ability to the user to change the radius of the selection sphere for 2D navigation in real-time, and thus, reach possibly unreachable atoms without ray casting to them.

Virtual buttons implementation

Each image in the panel, is a button that interacts with ray casting, and implements its own state machine to facilitate the interaction, based on the following interface:

```
public abstract class ButtonEvent : MonoBehaviour {
    public abstract void RayCastHoverOff();
    public abstract void RayCastHover();
    public abstract void RayCastHit();
}
```

The *RayCastHover()* function is called when a world space ray collides with the button, the *RayCastHoverOff()* is used to monitor if the ray is still colliding with the virtual button, and the *RayCastHit()* is used to signal that the button was hit.

There are three kinds of buttons. The first row of buttons, which are independent from each other, have four different states, one for each option, and their highlighted version. The state machine about the atom visualization button can be seen in the following image. The other two implement a similar one.

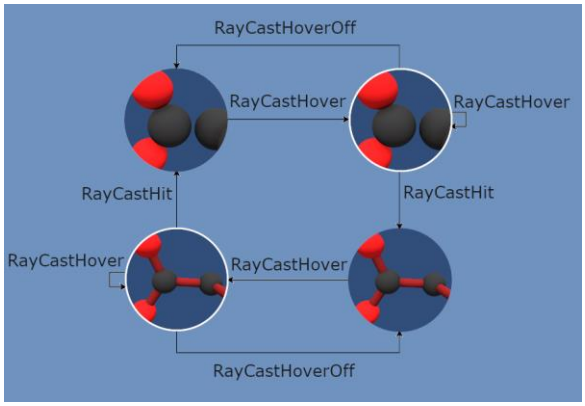


Figure 50: The state machine for the atom visualization virtual button

The second kind of button is the current mode buttons, which are not independent with each other, since the user can select any of them, in any order, and the other buttons must react to the change. An example of the state machine that they implement can be seen in the following image. The rest of the mode buttons implement a similar one.

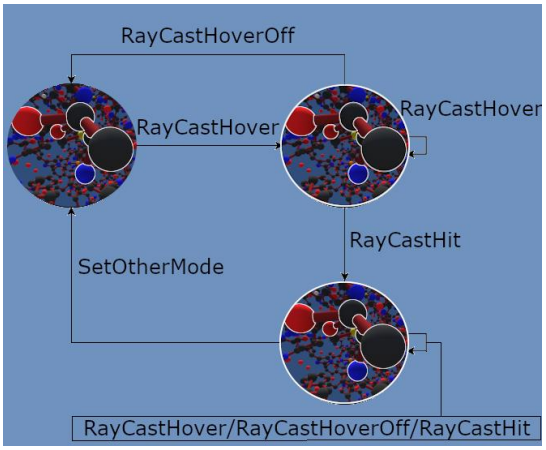


Figure 51: The state machine for the atom exploration mode virtual button

The third kind of button, are the buttons used to change the radius of the selection, whose state machine is very simple:

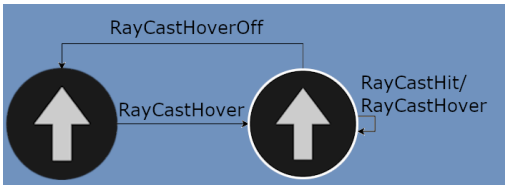


Figure 52: The state machine of the virtual button for increasing the radius of the navigation sphere

4.10 Model and World UI position

Since the application can read any input *.pdb* file, the initial position of the model inside the scene, as well as the initial position and rotation of the camera, must be calculated for each input model. To calculate the initial position of the model, we calculate the axis aligned bounding box of the input atoms, with size *(xsize, ysize, zsize)*, and place the model on top of the virtual floor, so that the bounding box barely touches the floor, using a small *epsilon* value.

The initial position of the camera is placed in front of the model looking towards the model in the *z* axis, at distance from the floor that is greater than $4 * {ysize}/{6}$, where $ysize$ is the size of the bounding box in the *y* axis. By using this distance value, the camera in the initial position, is slightly higher than the center of the bounding box.

The World UI panel is placed on the right of the model, by using the size that the bounding box has in the *x* axis, and adding a constant factor.

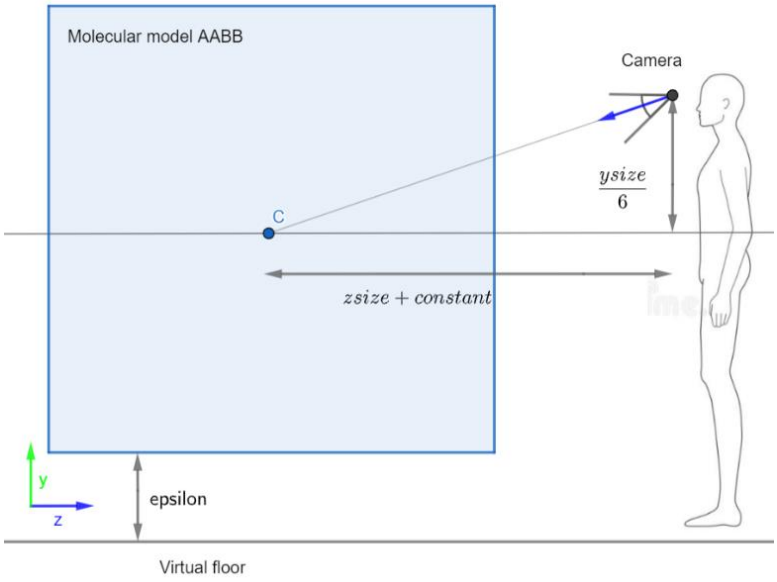


Figure 53: Camera and floor position

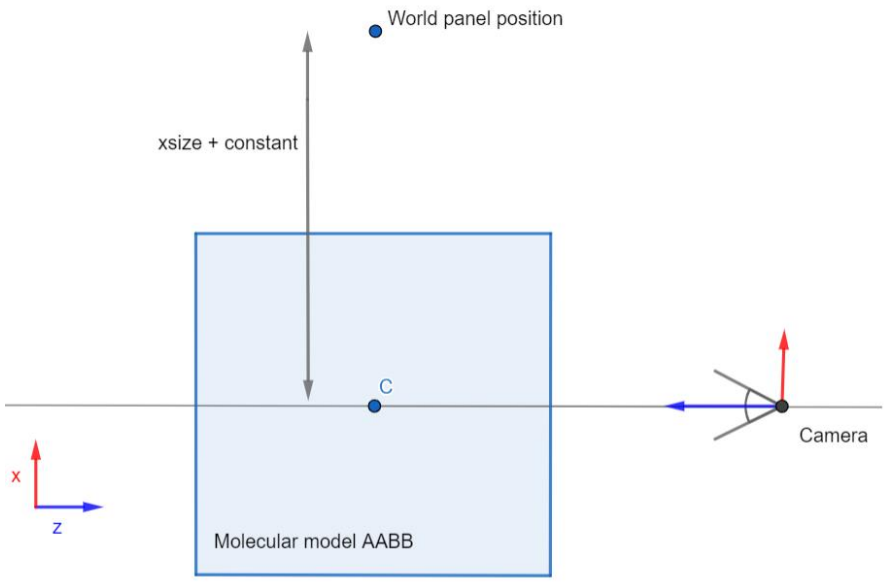


Figure 54: World panel position

The disadvantage of the above technique is that if the model is too large along the *y* axis, the user might get a weird feeling from the virtual floor being too far away. An alternative to this is, cap the distance that the virtual camera can have from the virtual floor on a certain distance value. This value has to be calculated of course by using a HMD.

4.11 HTC Vive Controller button mapping

In this chapter we are going to explore a possible mapping of operations into the physical buttons offered from the *HTC VIVE* controllers. We essentially need the following input for faster interaction:

Input	**Operation**
Selection button	Select an atom during ray casting, and when navigating between neighboring atoms and bonds, and for interacting with the world UI
2D directional input	Used to choose a direction when navigating between neighboring atoms and bonds
Mark button	Used to mark an atom in atom distances mode, and in torsion angle mode
Discard button	Used to exit the 2D selection, and start ray casting again
World UI button	Used to bring forward the world UI panel

Apart from the above operations, we can also map basic translate and rotate operations onto the controllers, so that the user can move the molecular model in space. For example:

Input	**Operation**
2D directional input	Move the whole model up, down, left, right in world space
2 Buttons	When the user has selected an atom, move the model towards or away from the user
Rotational input	Rotate the whole model

For the first table of interaction operations, we can use the right controller, and for the second table of movement operations we can use the left controller.

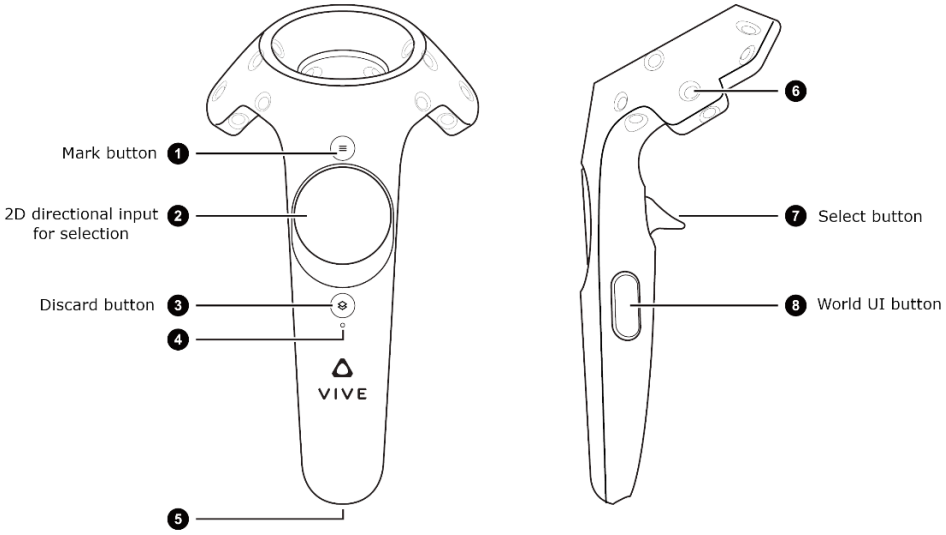


Figure 55: Possible mapping of interaction operation onto the right controller

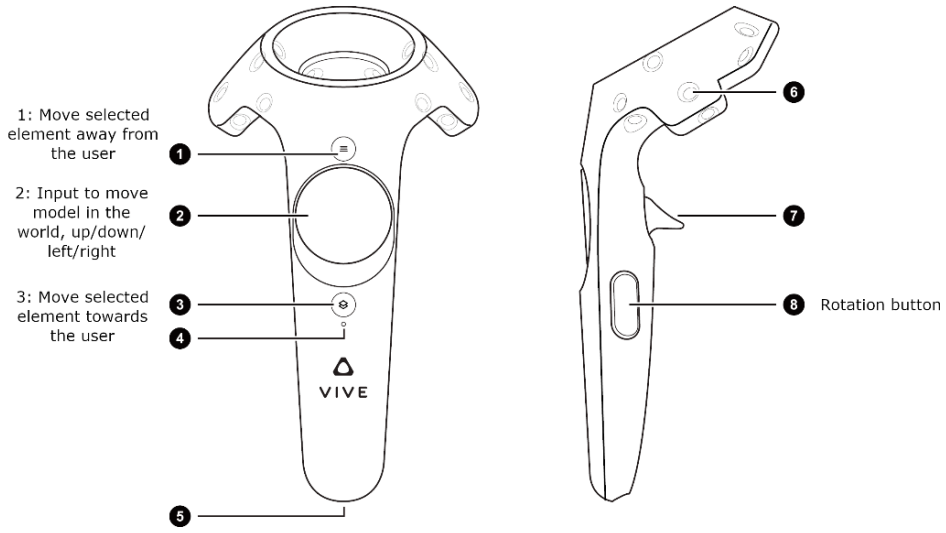


Figure 56: Possible mapping of model translation operation onto the left controller

As far as the rotation of the model is concerned, we can map a rotation button onto a physical button, and when this button is pushed, we can detect changes of rotation of the left controller, and apply these rotations onto the model. That way, the rotation of the model could be controlled from the user's hand rotations.

5. Usability experiments

In order to measure the usability of the 2D navigation technique, we were planning to conduct the following experiment. Create a simple scene, one that is not parsed from any *.pdb* model, and have users perform a specific task. For example, measure the distance between a pair of two red atoms, two green atoms, and two blue atoms. With similar logic, we can create tasks that involve bond angles and torsion angles. One version of the experiment would have users interacting only with ray casting, and another interacting with the 2D navigation scheme.

For the above experiment, the null hypothesis would be:

$$H_0 = Users\ need\ the\ same\ amount\ of\ time\ on\ average\ to\ complete\ all\ the\ tasks$$

If the hypothesis is found to be false, we could analyze the data gathered to determine which interaction method is more effective. Also, it would probably be more accurate, to perform a between-subjects experiment, since users usually get more accustomed with the controls over time.

To measure the usability of the world space UI, we could conduct the following experiment. Have users perform a specific task, that involves interaction with the UI. For example, start the application in exploring residues mode, and have the user measure the total number of chains present in the model. Then have the user, measure an atom distance, a bond angle, and finally a torsion angle. In one scenario the users would perform the above tasks with a static world space UI, and in the other scenario, a UI that can be brought forward above the geometry. The null hypothesis will again be H_0.

The data gathered and analyzed from the experiments could be, total time needed to complete all the tasks, the total number of times objects in the scene were visited, number of times 3D selection was used to select an object, number of times the user interacted with the UI, total amount of user movement inside the scene, etc.

In the end, this part was not completed, given the circumstances of the 2019-2 semester.

6. Conclusions and further work

Efficiently interacting with objects in a scene, is a highly challenging area in Virtual Reality, and HMD controller devices that are mainly designed around ray casting, cannot easily replace the keyboard and mouse. Furthermore, designing interfaces for visualizing information and changing parameters in the visualization, requires world space elements in the scene, which can in many cases be occluded. Molecular scenes pose an even greater challenge, due to the nature and complexity of the models.

Contributions

The initial idea for this work, was to implement a couple of simple interaction techniques, and perform a case study that involves real users, in order to examine the usability of each technique. For example, test systems like *UnityMol* [21], or *ChimeraX* [22]. However, due to unique circumstances, we were forced to diverge from the initial goal. As a result, the work was shifted into supporting more visualization models, for example the ball-and-stick, and adding more information extracted and visualized from the atoms and bonds.

This work provides an implementation for rendering high fidelity geometry objects (spheres and cylinders) by using ray casting, fully integrated within Unity, with support for physically based rendering, factory post processing effects and instancing.

An approach to navigation and selection in a molecular scene, that does not need to continuously use 3D selection methods, like virtual hand or ray casting, that can be easily expanded to all other kinds of objects, other than atoms and bonds. This navigation technique requires only one initial 3D selection by the user, and then the selection can be carried out with 2D controls, like trackpads.

A novel approach to a world space interaction canvas, that utilizes virtual buttons, and eases interaction when occluded.

Further work

There are several tasks left to made, mainly in the area of porting the application to a Virtual Reality headset. Everything was implemented with VR in mind, from rendering to interaction, as a result, the tasks remaining are: implementing the input interface between the HMD controllers and the application, checking that shaders can be used in Stereo instanced rendering mode, and configuring several other VR related options for example, movement sensitivity, filtering input, positioning, etc.

Apart from the above, the application can be extended in the following areas:

Expand in the molecular visualization area

Add support for reading other kinds of molecular model files, specifically files that allow the application to extract the bond information. Further expand the molecular information that can be visualized and add support for more visualization modes, apart from space-filling and ball-and-stick models.

Expand in the interaction area

Improve upon the initial 3D selection method used, which is currently simple ray casting, to incorporate a more elaborate method, tailor made for a molecular scene [18]. Add more features to the world space UI, for example, allow the user to store and discard information or set colors or highlighting to specific objects in the scene. Optimally position the model, the camera, and the world UI, independently of the

input model, by efficiently calculating the object-oriented bounding box of the model, instead of the axis aligned bounding box.

Expand in the molecular rendering area

Add feature-complete sphere and cylinder impostors, with casting and receiving shadows, UV coordinates, and texture mapping, or further expand impostor geometry to ray-casted atom hyperballs [23].

Apart from the above, it is vital to conduct a user study to determine the usability of the interaction methods created.

The complete project can be found here: https://github.com/kwstanths/MRend

Bibliography

[1] J. Liang and M. Green, "JDCAD: A highly Interactive 3D Modeling System," *Computer & Graphics,* vol. 18, no. 4, pp. 499-506, 1994.

[2] A. Olwal and S. Feiner, "The Flexible Pointer: An Interaction technique for Augmented and Virtual Reality," in *User Interface Software and Technology*, 2003.

[3] J. P. e. al, "Image Plane Interaction Techniques in 3D Immersive Environments," in *Interactive 3D Graphics*, 1997.

[4] F. Argelaguet and C. Andujar, "Efficient 3D Pointing Selection in Cluttered Virtual Environments," *IEEE Computer Graphics and Applications,* vol. 29, no. 6, pp. 34-43, 2009.

[5] L. Pauling, "The Nature of the Chemical Bond (2nd ed.)," Cornell University Press, 1945.

[6] J. M. Berg, J. L. Tymoczko and L. and Stryer, Biochemistry, 5th edition, New York, 2002.

[7] "Unity Documentation, Providing vertex data to vertex programs," Unity, 2019. [Online]. Available: https://docs.unity3d.com/Manual/SL-VertexProgramInputs.html. [Accessed 21 March 2020].

[8] "Unity Documentation, Shader semantics," Unity, [Online]. Available: https://docs.unity3d.com/Manual/SL-ShaderSemantics.html. [Accessed 21 3 2020].

[9] "Unity Documentation, SubShader tags," Unity, [Online]. Available: https://docs.unity3d.com/Manual/SL-SubShaderTags.html. [Accessed 22 3 2020].

[10] "Unity Documentation, Pass Tags," Unity, [Online]. Available: https://docs.unity3d.com/Manual/SL-PassTags.html. [Accessed 22 3 2020].

[11] "Unity Documentation, Deferred Shading," Unity, [Online]. Available: https://docs.unity3d.com/Manual/RenderTech-DeferredShading.html. [Accessed 22 3 2020].

[12] "Unity Documentation, Draw call batching," Unity, [Online]. Available: https://docs.unity3d.com/Manual/DrawCallBatching.html. [Accessed 22 3 2020].

[13] "Unity Documentation, GPU instancing," Unity, [Online]. Available: https://docs.unity3d.com/Manual/GPUInstancing.html. [Accessed 22 3 2020].

[14] "Unity Documentation, Ambient Occlusion," Unity, [Online]. Available: https://docs.unity3d.com/Packages/com.unity.postprocessing@2.0/manual/Ambient-Occlusion.html. [Accessed 23 3 2020].

[15] "Unity Documentation," Unity, [Online]. Available: https://docs.unity3d.com/ScriptReference/Rendering.RenderQueue.html. [Accessed 7 5 2020].

[16] "Unity Documentation, Single Pass Stereo Rendering," Unity, [Online]. Available: https://docs.unity3d.com/Manual/SinglePassStereoRendering.html. [Accessed 24 5 2020].

[17] "Unity Documentation, Single Pass Instancing," Unity, [Online]. Available: https://docs.unity3d.com/Manual/SinglePassInstancing.html. [Accessed 24 5 2020].

[18] F. Argelaguet and C. Andujar, "A survey of 3D object selection techniques for virtual environments," *Computer Graphics,* vol. 37, pp. 121-136, 2013.

[19] "Compendium of Chemical Terminology, 2nd ed, "Gold Book"," IUPAC, 1997.

[20] E. Anslyn and D. Dougherty, Modern Physical Organic Chemistry, 2006.

[21] Baaden-Chavent-Martine, "UnityMol," [Online]. Available: http://www.baaden.ibpc.fr/umol/. [Accessed 15 6 2020].

[22] USFC, "ChimeraX," [Online]. Available: https://www.cgl.ucsf.edu/chimerax/. [Accessed 15 6 2020].

[23] M. Chavent, A. Vanel, A. Tek, B. Levy, S. Robert, B. Raffin and M. Baaden, "GPU-accelerated atom and dynamic bond visualization using hyperballs: A unified algorithm for balls, sticks, and hyperboloids," *Journal of Computational Chemistry,* vol. 32, no. 13, p. 2924-2935, 2011.

Appendix

A: Ray-Sphere intersection

```
void ImpostorSphere(float3 fragment_position_worldspace, float
       sphere_radius, inout float3 position_worldspace, inout float3
       normal_worldspace, bool back_face = false)
{

    float3 fragment_pos = fragment_position_worldspace;
    /* World space position of the center of the sphere, object space
       (0,0,0) */
    float3 sphere_center = mul(UNITY_MATRIX_M, float4(0.0, 0.0, 0.0,
1.0)).xyz;

    /* A is the origin of the ray */
    float3 A = _WorldSpaceCameraPos.xyz;
    /* B is the direction of the ray */
    float3 B = normalize(fragment_pos - _WorldSpaceCameraPos.xyz);
    /* C is the sphere center */
    float3 C = sphere_center;

    /* Solve the ray, sphere intersection problem */
    float a = dot(B, B);
    float b = 2.0f * dot(B, A - C);
    float c = dot(A - C, A - C) - (sphere_radius * sphere_radius);

    /* Calculate delta */
    float delta = (b * b) - (4.0f * a * c);
    /* Clip fragment if delta is negative */
    clip(delta);

    /* If not, calculate world space position of intersection point */
    delta = sqrt(delta);
    float t1 = (0.5f) * (-b + delta) / a;
    float t2 = (0.5f) * (-b - delta) / a;
    /* Get the closest point */
    float t;
    if (!back_face) t = min(t1, t2);
    else t = max(t1, t2);

    position_worldspace = A + B * t;
    normal_worldspace = normalize(position_worldspace - sphere_center);
}
```

B: Ray-Cylinder intersection

```
void ImpostorCylinder2(float3 fragment_position_worldspace, float3
cylinder_direction_worldspace, float cylinder_radius,
    float cylinder_height, inout float3 position_worldspace, inout float3
normal_worldspace)
{
    /* Calculate geometry cylinder data, assume cylinder center is at
(0,0,0) object space */
    float3 C = mul(UNITY_MATRIX_M, float4(0.0, 0.0, 0.0, 1.0)).xyz;
    float3 e = cylinder_direction_worldspace;
    float r = cylinder_radius;
```

```
    /* Calculate ray */
    float3 P = _WorldSpaceCameraPos.xyz;
    float3 v = normalize(fragment_position_worldspace -
_WorldSpaceCameraPos.xyz);

    /* Precalculate some values used more than twice */
    float CPe = dot(C - P, e);
    float CPv = dot(C - P, v);
    float ev = dot(e, v);

    /* Calculate lambda for the point A that is closest on the cylinder axis
*/
    float lambda = (ev / (pow(ev, 2) - 1)) * (CPe - CPv / ev);
    /* Calulcate point A */
    float3 A = P + lambda * v;
    /* Calculate the distance of that point to the cylinder axis */
    float d = GetDistanceFromPointToLine(A, C, e);
    /* If negative, discard fragment */
    clip(r - d);
    /* Calculate the intersection point lambda value */
    float l_prime = lambda - sqrt((pow(r, 2) - pow(d, 2)) / (1 -
pow(ev,2)));
    /* Calulcate intersection point */
    position_worldspace = P + l_prime * v;

    /*
        Calculate normal of that intersection point
        project that point on the cylinder axis, clip the projection based
on the height required
        and then calculate normal using the projection to find the
corresponding point on the axis
    */
    float projection = dot(position_worldspace - C, e);
    clip(projection);
    clip(cylinder_height - projection);

    normal_worldspace = normalize(position_worldspace - (C + projection *
e));
}
```

C: Sphere impostor Unity deferred and forward shader

```
Shader "Custom/ImpostorSphere"
{
    Properties{
        _Albedo("Albedo", Color) = (1, 0, 0.8, 0)
        _RadiusAndShading("_RadiusAndShading", Color) = (0.07, 0.7, 0, 0)
    }

    SubShader {
        Tags { "Queue" = "Geometry" }

        Pass {
            /* Base forward rendering shader, executed when the rendering
mode is set to forward, and with the directional light as input */
            Tags { "LightMode" = "ForwardBase" }
```

```
            /* This shader will be used for transparent rendering, with
specific rendering order */
            Blend SrcAlpha OneMinusSrcAlpha
            ZWrite Off

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_instancing

            #include "UnityCG.cginc"
            #include "Lightning.cginc"
            #include "Impostor.cginc"
            #include "AutoLight.cginc"

            #define BOX_CORRECTION 1.5

            /* Provided by Unity */
            uniform float4 _LightColor0;

            struct appdata {
                /* Instance ID */
                UNITY_VERTEX_INPUT_INSTANCE_ID
                /* Object space position */
                float4 vertex : POSITION;
            };
            struct v2f {
                UNITY_VERTEX_INPUT_INSTANCE_ID
                /* clip space position */
                float4 pos : SV_POSITION;
                /* view space position */
                float4 view_pos : TEXCOORD0;
                /* Single pass instanced rendering */
                UNITY_VERTEX_OUTPUT_STEREO
            };

            /* Unpack extra instance properties */
            UNITY_INSTANCING_BUFFER_START(Props)
                UNITY_DEFINE_INSTANCED_PROP(float4, _Albedo)
                UNITY_DEFINE_INSTANCED_PROP(float4, _RadiusAndShading)
            UNITY_INSTANCING_BUFFER_END(Props)

            v2f vert(appdata input)
            {
                v2f output;
                UNITY_SETUP_INSTANCE_ID(input);
                UNITY_INITIALIZE_OUTPUT(v2f, output);
                UNITY_TRANSFER_INSTANCE_ID(input, output);
                UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(output);

                float radius = UNITY_ACCESS_INSTANCED_PROP(Props,
_RadiusAndShading).r;

                /* Transform standard quad geometry to face the camera */
                /* Multiply the width of the quad with the box correction */
```

```
                /* Multiply with 2 sinxe the standard quad geometry goes
from -0.5 to 0.5 and we want the standard sphere to have radius 1 */
                output.view_pos = mul(UNITY_MATRIX_MV, float4(0.0, 0.0, 0.0,
1.0)) + BOX_CORRECTION * float4(input.vertex.x, input.vertex.y, 0.0, 0.0) *
2.0f * float4(radius, radius, 1.0, 1.0);
                output.pos = mul(UNITY_MATRIX_P, output.view_pos);

                return output;
            }

            float4 frag(v2f input, out float outDepth : SV_Depth) : COLOR
            {
                /* Set up instance id */
                UNITY_SETUP_INSTANCE_ID(input);
                UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(input);

                float4 radius_and_shading =
UNITY_ACCESS_INSTANCED_PROP(Props, _RadiusAndShading);
                float radius = radius_and_shading.r;
                float ambient_factor = radius_and_shading.g;

                /* Compute real fragment world position and normal */
                float3 normal_world, position_world;
                ImpostorSphere(mul(UNITY_MATRIX_I_V, input.view_pos),
radius, position_world, normal_world);

                /* Calculate depth */
                float4 clip = mul(UNITY_MATRIX_VP, float4(position_world,
1.0f));
                float z_value = clip.z / clip.w;
                outDepth = z_value;

                float3 view_direction = normalize(position_world -
_WorldSpaceCameraPos.xyz);

                /* Phong shading */
                DirectionalLight light;
                light.direction = -_WorldSpaceLightPos0.xyz;
                light.ambient_factor = ambient_factor;
                light.diffuse_color = _LightColor0;

                float4 albedo = UNITY_ACCESS_INSTANCED_PROP(Props, _Albedo);

                float3 color = DirectionalLightColor(light, normal_world,
view_direction, albedo.xyz);

                return half4(color, 0.35);
            }
            ENDCG
        }

        Pass {
            /* Base forward rendering shader, executed when the rendering
mode is set to forward, and with a point light as input */
            Tags { "LightMode" = "ForwardAdd" }

            Blend One One
```

```
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_instancing

            #include "Lightning.cginc"
            #include "Impostor.cginc"
            #include "UnityCG.cginc"

            #define BOX_CORRECTION 1.5

            /* Provided by Unity */
            uniform float4 _LightColor0;

            struct appdata {
                /* Instance ID */
                UNITY_VERTEX_INPUT_INSTANCE_ID
                /* Object space position */
                float4 vertex : POSITION;
            };
            struct v2f {
                UNITY_VERTEX_INPUT_INSTANCE_ID
                /* clip space position */
                float4 pos : SV_POSITION;
                /* view space position */
                float4 view_pos : TEXCOORD0;
                /* Single pass instanced rendering */
                UNITY_VERTEX_OUTPUT_STEREO
            };

            UNITY_INSTANCING_BUFFER_START(Props)
                UNITY_DEFINE_INSTANCED_PROP(float4, _Albedo)
                UNITY_DEFINE_INSTANCED_PROP(float4, _RadiusAndShading)
                UNITY_INSTANCING_BUFFER_END(Props)

            v2f vert(appdata input)
            {
                v2f output;
                UNITY_SETUP_INSTANCE_ID(input);
                UNITY_INITIALIZE_OUTPUT(v2f, output);
                UNITY_TRANSFER_INSTANCE_ID(input, output);
                UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(output);

                float radius = UNITY_ACCESS_INSTANCED_PROP(Props,
_RadiusAndShading).r;

                output.view_pos = mul(UNITY_MATRIX_MV, float4(0.0, 0.0, 0.0,
1.0)) + BOX_CORRECTION * float4(input.vertex.x, input.vertex.y, 0.0, 0.0) *
2.0f * float4(radius, radius, 1.0, 1.0);
                output.pos = mul(UNITY_MATRIX_P, output.view_pos);

                return output;
            }

            float4 frag(v2f input, out float outDepth : SV_Depth) : COLOR
```

```
            {
                /* Set up instance id */
                UNITY_SETUP_INSTANCE_ID(input);
                UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(input);

                float4 radius_and_shading =
UNITY_ACCESS_INSTANCED_PROP(Props, _RadiusAndShading);
                float radius = radius_and_shading.r;
                float ambient_factor = radius_and_shading.g;

                float3 normal_world, position_world;
                ImpostorSphere(mul(UNITY_MATRIX_I_V, input.view_pos),
radius, position_world, normal_world);

                float4 clip = mul(UNITY_MATRIX_VP, float4(position_world,
1.0f));
                float z_value = clip.z / clip.w;
                outDepth = z_value;

                float3 view_direction = normalize(position_world -
_WorldSpaceCameraPos.xyz);

                PointLight light;
                light.position = _WorldSpaceLightPos0.xyz;
                light.ambient_factor = ambient_factor;
                light.diffuse_color = _LightColor0;

                float4 albedo = UNITY_ACCESS_INSTANCED_PROP(Props, _Albedo);

                float3 color = PointLightColor(light, position_world,
normal_world, view_direction, albedo.xyz);

                return half4(color, 0.35);
            }
            ENDCG
        }

        Pass {
            /* Deferred rendering shader, executed when the rendering mode
is set to deferred */
            Tags { "LightMode" = "Deferred" }
            CGPROGRAM

            #pragma target 3.0
            /* Exclude GPU that don't support Multi Target Rendering */
            #pragma exclude_renderers nomrt

            /* Define the vertex and fragment shader programs */
            #pragma vertex vert
            #pragma fragment frag
            /* Add multi compiling support for instancing */
            #pragma multi_compile_instancing

            #include "UnityCG.cginc"
            #include "Impostor.cginc"
            #include "UnityPBSLighting.cginc"
```

```
            #define BOX_CORRECTION 1.5

            struct appdata {
                /* Instance ID */
                UNITY_VERTEX_INPUT_INSTANCE_ID
                /* Object space position */
                float4 vertex : POSITION;
            };
            struct v2f {
                UNITY_VERTEX_INPUT_INSTANCE_ID
                /* clip space position */
                float4 pos : SV_POSITION;
                /* view space position */
                float4 view_pos : TEXCOORD0;
                /* Single pass instanced rendering */
                UNITY_VERTEX_OUTPUT_STEREO
            };
            struct fragment_output
            {
                half4 diffuse : SV_Target0;
                half4 specular : SV_Target1;
                half4 normal_world : SV_Target2;
                half4 emission : SV_Target3;
            };

            /* Unpack extra instance properties */
            UNITY_INSTANCING_BUFFER_START(Props)
                UNITY_DEFINE_INSTANCED_PROP(float4, _Albedo)
                UNITY_DEFINE_INSTANCED_PROP(float4, _RadiusAndShading)
            UNITY_INSTANCING_BUFFER_END(Props)

            v2f vert(appdata input)
            {
               v2f output;
               UNITY_SETUP_INSTANCE_ID(input);
               UNITY_INITIALIZE_OUTPUT(v2f, output);
               UNITY_TRANSFER_INSTANCE_ID(input, output);
               UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(output);

               float radius = UNITY_ACCESS_INSTANCED_PROP(Props,
_RadiusAndShading);

               output.view_pos = mul(UNITY_MATRIX_MV, float4(0.0, 0.0, 0.0,
1.0)) + BOX_CORRECTION * float4(input.vertex.x, input.vertex.y, 0.0, 0.0) *
2.0f * float4(radius, radius, 1.0, 1.0);
               output.pos = mul(UNITY_MATRIX_P, output.view_pos);

               return output;
            }

            fragment_output frag(v2f input, out float outDepth : SV_Depth) :
COLOR
            {
                /* Set up instance id */
                UNITY_SETUP_INSTANCE_ID(input);
                UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(input);
```

```
                float4 radius_and_shading =
UNITY_ACCESS_INSTANCED_PROP(Props, _RadiusAndShading);
                float radius = radius_and_shading.r;
                float ambient_factor = radius_and_shading.g;
                float metallic = radius_and_shading.b;
                float gloss = radius_and_shading.a;

                /* Compute real fragment world position and normal */
                float3 normal_world, position_world;
                ImpostorSphere(mul(UNITY_MATRIX_I_V, input.view_pos),
radius, position_world, normal_world);

                /* Calculate depth */
                float4 clip = mul(UNITY_MATRIX_VP, float4(position_world,
1.0f));
                float z_value = clip.z / clip.w;
                outDepth = z_value;

                /* Calculate albedo for this instance */
                float4 albedo = UNITY_ACCESS_INSTANCED_PROP(Props, _Albedo);
                float is_highlighted = albedo.w;

                /* Calculate diffuse and specular component from using
Unity's Physically based rendering pipeline */
                half3 specular;
                half specularMonochrome;
                half3 diffuseColor =
DiffuseAndSpecularFromMetallic(albedo.xyz, metallic, specular,
specularMonochrome);

                /* Set output parameters */
                fragment_output o;
                o.diffuse = float4(diffuseColor, 1);
                o.specular = half4(specular, gloss);
                o.normal_world.xyz = normal_world * 0.5f + 0.5f;
                o.normal_world.w = is_highlighted;
                o.emission.xyz = ambient_factor * diffuseColor;
                return o;
            }
            ENDCG
        }
    }

    Fallback "Diffuse"
}
```

D: Phong shading functions

```
#ifndef __Lightning_cginc__
#define __Lightning_cginc__

/* Phong shading parameters used in forward rendering */
static float _Shininess = 32;
static float _SpecularIntensity = 0.2f;

struct DirectionalLight {
    float3 direction;
    float ambient_factor;
```

```
    float3 diffuse_color;
};

struct PointLight {
    float3 position;
    float3 ambient_factor;
    float3 diffuse_color;
};

/* Calculate linear attenuation between the position of the fragment and the
light */
float Attenuation(float3 fragment_position, float3 light_position) {
    float3 vert = light_position - fragment_position;
    float distance_inv = 1.0 / length(vert);
    return lerp(0.0, 1.0, distance_inv);
}

/* Calculate directional light color contribution */
float3 DirectionalLightColor(DirectionalLight light, float3 fragment_normal,
float3 view_direction, float3 fragment_color) {
    /* Ambient component */
    float3 light_ambient = light.ambient_factor * (fragment_color *
light.diffuse_color);

    /* Diffuse component */
    float3 light_direction_inv = normalize(-light.direction);
    float light_diffuse_strength = dot(fragment_normal,
light_direction_inv);
    float3 light_diffuse = max(light_diffuse_strength, 0.0f) *
(fragment_color * light.diffuse_color);

    float3 light_specular = float3(0, 0, 0);
    if (light_diffuse_strength > 0) {
        /* Specular component */
        /* Find the reflected vector from the light towards the surface
normal */
        float3 light_reflect_vector = reflect(light_direction_inv,
fragment_normal);
        float light_specular_strength = pow(max(dot(view_direction,
light_reflect_vector), 0.0), _Shininess);
        light_specular = light.diffuse_color * light_specular_strength *
_SpecularIntensity;
    }

    return light_ambient + light_diffuse + light_specular;
}

float3 PointLightColor(PointLight light, float3 fragment_position, float3
fragment_normal, float3 view_direction, float3 fragment_color) {
    /* Ambient component */
    float3 light_ambient = light.ambient_factor * (fragment_color *
light.diffuse_color);

    /* Calculate diffuse component */
    float3 light_direction_inv = normalize(light.position -
fragment_position);
```

```
    float light_diffuse_strength = dot(fragment_normal,
light_direction_inv);
    float3 light_diffuse = max(light_diffuse_strength, 0.0f) *
(fragment_color * light.diffuse_color);

    float3 light_specular = float3(0, 0, 0);
    if (light_diffuse_strength > 0) {
        /* Specular component */
        /* Find the reflected vector from the light towards the surface
normal */
        float3 light_reflect_vector = reflect(light_direction_inv,
fragment_normal);
        float light_specular_strength = pow(max(dot(view_direction,
light_reflect_vector), 0.0), _Shininess);
        light_specular = light.diffuse_color * light_specular_strength *
_SpecularIntensity;
    }

    return light_ambient + Attenuation(fragment_position, light.position) *
(light_diffuse + light_specular);
}

#endif
```

E: Border highlighting post processing effect Unity shader

```
Shader "Hidden/Custom/Outline"
{
    HLSLINCLUDE
    #include
"Packages/com.unity.postprocessing/PostProcessing/Shaders/StdLib.hlsl"

    /* Currently drawn texture */
    TEXTURE2D_SAMPLER2D(_MainTex, sampler_MainTex);
    /* Thickness parameter */
    int _Thickness;
    /* The Gbuffer render target with the highlighted information */
    sampler2D _CameraGBufferTexture2;
    half4 _CameraGBufferTexture2_ST;
    /* The size inforamtion for the above texture */
    float4 _CameraGBufferTexture2_TexelSize;

    float4 GetHighlightColor(float texture_value) {
        if (texture_value > 0.7) return float4(1, 1, 1, 1);
        else if (texture_value > 0.4) return float4(0, 0, 1, 1);
        else return float4(0, 0.9, 0, 1);
    }

    float4 SampleTexture(float2 uv) {
        return tex2D(_CameraGBufferTexture2,
UnityStereoScreenSpaceUVAdjust(uv, _CameraGBufferTexture2_ST));
    }

    float4 Frag(VaryingsDefault i) : SV_Target
    {
        /* Get size information */
        float2 tex_size = _CameraGBufferTexture2_TexelSize.xy;
        /* Get value for current fragment */
```

```
        float selected_value = SampleTexture(i.texcoord).w;
        bool is_border = false;
        /* If this fragment is highlighted, it could belong to the border */
        if (selected_value > 0) {
            /* Sample a number of fragments around, and if any of them is
not highlighted
                then this fragment belongs to the border */
            [loop]
            for (int x = -_Thickness; x <= +_Thickness; x++) {
                [loop]
                for (int y = -_Thickness; y <= +_Thickness; y++) {
                    if (x == 0 && y == 0) {
                        continue;
                    }
                    float2 offset = float2(x, y) * tex_size;
                    if (SampleTexture(i.texcoord + offset).w !=
selected_value) {
                        is_border = true;
                    }
                }
            }
        }

        /* If it's border return color of outline */
        if (is_border) return GetHighlightColor(selected_value);

        /* Otherwise, return previous color */
        return SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, i.texcoord);
    }

    ENDHLSL

    SubShader
    {
        Cull Off ZWrite Off ZTest Always

        Pass
        {
            HLSLPROGRAM

            #pragma vertex VertDefault
            #pragma fragment Frag

            ENDHLSL
        }
    }
}
```

F: Stereo instancing Unity built-in matrix variables

UnityCG.cginc:137

```
// Tranforms position from world to homogenous space
inline float4 UnityWorldToClipPos( in float3 pos )
{
    return mul(UNITY_MATRIX_VP, float4(pos, 1.0));
}
```

UnityShaderVariables.cginc:16

```
#if defined(USING_STEREO_MATRICES)
    #define glstate_matrix_projection
unity_StereoMatrixP[unity_StereoEyeIndex]
    #define unity_MatrixV unity_StereoMatrixV[unity_StereoEyeIndex]
    #define unity_MatrixInvV unity_StereoMatrixInvV[unity_StereoEyeIndex]
    #define unity_MatrixVP unity_StereoMatrixVP[unity_StereoEyeIndex]

    #define unity_CameraProjection
unity_StereoCameraProjection[unity_StereoEyeIndex]
    #define unity_CameraInvProjection
unity_StereoCameraInvProjection[unity_StereoEyeIndex]
    #define unity_WorldToCamera
unity_StereoWorldToCamera[unity_StereoEyeIndex]
    #define unity_CameraToWorld
unity_StereoCameraToWorld[unity_StereoEyeIndex]
    #define _WorldSpaceCameraPos
unity_StereoWorldSpaceCameraPos[unity_StereoEyeIndex]
#endif

#define UNITY_MATRIX_P glstate_matrix_projection
#define UNITY_MATRIX_V unity_MatrixV
#define UNITY_MATRIX_I_V unity_MatrixInvV
#define UNITY_MATRIX_VP unity_MatrixVP
#define UNITY_MATRIX_M unity_ObjectToWorld
```