Small Project Report

Athos Van Kralingen (6037062)
a.l.vankralingen@uu.nl
Utrecht University
Utrecht, Netherlands

Jake Greenshields (1600702)
j.w.h.greenshields@students.uu.nl
Utrecht University
Utrecht, Netherlands

Lorenzo Marsicano (2024578)
l.marsicano@students.uu.nl
Utrecht University
Utrecht, Netherlands

1 INTRODUCTION

This report summarizes the work that has been done as part of the Small Project for GMT to provide a starting point for other students to continue what we started.

As dissemination for their master theses last year, Xander Hermans and William van der Scheer both made contributions to the voxel-based game template, WrldTmpl8, created by Jacco Bikker. Hermans created and implemented a way to apply Reservoir-based Spatio-Temporal Importance Resampling (ReSTIR) to a voxelized world [2], so that it can handle larger numbers of emissive voxels simultaneously.

Van der Scheer [4] delivered a fluid simulation method based on cellular automata that allowed for the simulation of fluids of different viscosity in real-time.

The desire is to make the WrldTmpl8 project be a good starting place for novice programmers to make their first games in C++ with various tools in place and we think that both these mentioned projects could enrich this project significantly. However, these projects were not made with ease of use, re-usability or combination with the other project in mind. Additionally, despite having a fluid simulation, transparency is current not possible as a colour component for each voxel. This means the rendering of water, one of the expected most common usecases of the fluid simulation, lacks the realistic aspect of being able to look through it. Finally, any part of a game world currently has to be added through code. While this can be sufficient, iteration times can be significantly sped up if there was a way of editing the contents of the world through a world editor. Therefore, next to the previous aspects, we will try to introduce a world editor in which voxels can be edited in a convenient way.

In this report, we try to outline the work we have done to make this possible, but also the difficulties we faced in attempting to combine the fluid simulation and ReSTIR projects and difficulties we encountered while working with WrldTmpl8 in general.

For the Small Project, the following results were achieved:

- we merged the CAPE code written by William van der Scheer and the ReSTIR implementation by Xander Hermans into the main Voxel Template codebase;
- we implemented a Voxel World Editor into the template to facilitate the creation of levels and games;
- we added the possibility to specify an alpha value for each voxel, giving to the users the possibility to have translucent objects;
- we extended the voxel size from 16 bits to 32 bits. This allowed us to use more bits for the emissive strength and to add the alpha value to the payload;
- we created a demo level to show off ReSTIR capabilities with the added transparency;

2 MERGING CAPE AND RESTIR CODE

The first step of the project was to identify the differences between the CAPE and the ReSTIR code bases.
A quick diff between the two projects allowed us to perform a merge of the biggest differences, for example extra kernel arguments or the initialization functions for the water simulation.
The class that we had to take bigger care in the merging process was the *World* class

2.1 Refactoring the ReSTIR code into the template

One of the aim of the small project was to have ReSTIR always on by default in every project.
To do that we moved the initialization code into the **World** class.
The default parameters and the reservoir buffers are initialized from the Template *main* entry point after instantiating the **World** class.
Following that, the **Game** class is initialized, which is responsible to populate the **World** light buffer, either by adding lights manually using **World::Set** or by calling **World::FindLightsInWorld** to automatically find all voxels (and bricks) that have an emissive value in the world.
If the game does not initialize the light buffers that are sent to the GPU, the template does it instead by using an empty light vector.
At runtime, the lights are moved and sampled the same way as it was before, but the function calls that take care of doing so have been moved to the **World::Update** function instead of the **Game::Update**

2.2 The LightManager class

The ReSTIR code added a LightManager class that took care of the setup of the Lights buffer as well as of updating the moving lights.
We noticed that we were having invalid deletes of the lights buffer and uses after it being deleted at runtime, so we decided to completely eliminate the class and move the needed members and function into the World class.
Furthermore, some missing functions (namely **AddLight** and **RemoveLight**) were added, which were also useful for the world editor.

2.3 The FluidSimulator class

To isolate CAPE code and make it more usable, we created a FluidSimulator class that act as a wrapper around the calls to the CAPE library.
By including the FluidSimulator class into the game, we can easily set water blocks with **FluidSimulator::SetMaterialBlock** and update the simulation at every timestep, making it easier to add a body of water to the project.

3 RESOLVING PROBLEMS FOR CAPE

While merging CAPE onto the the ReSTIR codebase we started noticing some rendering issues. After digging deeper into it we decided to not include the fluid simulation in our demo, as we could not integrate it correctly with the current architecture.

The snippet shown in Listing 1 from **World::Clear** was initializing each brick in the grid with its own identifier and manually setting it as *empty* so that there would be no need to create bricks at runtime. Additionally, each brick was fully allocated up front at startup, which all meant that the code could make the assumption every brick it would edit voxels of would have an offset into the *brick* buffer. This together breaks the compression scheme, as each brick would be required to have memory allocated in the *brick* buffer at all times. Another difference that related to this was the **World::Set** method in the ReSTIR project: that version would split a solid brick if needed, find room in the *brick* buffer for this brick that is no longer solid and set the individual voxel at the right location. For CAPE this functionality would not change anything, as the brick would already be non-solid at all times.

Furthermore, the **OptimizeBricks** functionality had essentially been disabled. Normally this function is used to transform bricks that are considered non-solid to solid ones if all the voxels in it have the same value, freeing up space in the *brick* buffer by only storing a single value for an entire 8x8x8.

Finally, we found that **GRIDSTEP** macro in *cl/trace.cl* shown in Listing 3 had also been rewritten as the one shown in Listing 2 in CAPE codebase. Normally, the `(o & 1)` check is performed to see if the brick is solid and therefore contains a uniform value for all voxels within, but from this it can be seen that the CAPE project repurposed this so that any solid brick would be considered to be completely empty. Without this change, the initialisation it performed in the **World::Clear** function would cause the renderer to treat the identifiers stored in each *grid* cell as a colour.

These measures seem to have been taken so that the physics engine itself, which runs on the GPU, would not have to deal with converting solid bricks to non-solid bricks on the GPU. Doing this can be difficult as it would require additional allocation on the GPU for all the possible bricks that could be split, but we will discuss this in future work in more detail.

To summarise, there were two options to integrate CAPE into the project:

(1) Make every brick non-solid and pre-allocate all bricks in advance as done in CAPE.
(2) Create a method to split solid bricks into non-solid bricks on the GPU and rewrite part of CAPE so that it does this when necessary.

We decided that option 1 is never a good option, as the memory requirements become huge and the current compression scheme is no longer doing anything. The second option involved several design challenges and we ended up dedicating the time to implementing translucency instead.

Listing 1: Problematic code in World::Clear

```
1  for (int i = 0; i < GRIDSIZE; i++)
2    grid[i] = (i << 1) | 0; //zero identifier
```

3.1 Future Work

While we did not work on an implementation, we did draft a design for making CAPE work by giving the GPU the ability to transform a solid brick into a non-solid one. This would work as follows:

(1) Pre-allocate a buffer on the GPU of a size that seems reasonable as for the number of bricks the CAPE simulation may touch to store the new non-solid bricks in. Note that this limit will have to be checked to not be exceeded, it will simply have to be a limitation for this to work if VRAM is limited.
(2) Pre-allocate a buffer of brick coordinates that follows the same size limitation as the buffer in step 1.
(3) Perform a pass in which is determined if CAPE requires editing of an individual voxel and check if the brick of this voxel is currently solid.
(4) If it is non-solid, we can edit as normal. If it is solid, we go to step 3.
(5) Atomically get a unique index for the buffer from step 2 and write the location of the brick as a single index into it. Do this for all bricks that need to be made non-solid.
(6) In a next pass, eliminate all the duplicates in the buffer from step 2. Then run a pass that for each brick location, writes that brick as individual voxels to the buffer allocated in step 1.
(7) Run the full CAPE simulation step and afterwards write the results back to main memory on the CPU.

Listing 2: Tracing code from the CAPE codebase

```
1  #define GRIDSTEP(exitX)
2    if (!--steps) break;
3    if (o & 1)
4    {
5      const float4 tm_ = tm;
6      const uint4 p4 = convert_uint4( A + V * (t *= 8) );
7      ...
```

Listing 3: Tracing code from the ReSTIR codebase

```
1  #define GRIDSTEP(exitX)
2    if (!--steps) break;
3    if (o != 0) if ((o & 1) == 0) { *dist = (t + to) * 8.0f
       , *side = last; return o >> 1; }
4    else
5    {
6      const float4 tm_ = tm;
7      const uint4 p4 = convert_uint4( A + V * (t *= 8) );
8          ...
```

An alternative to the buffer mentioned in step 2 is a buffer of 1 bit per brick in the entire world that is set to 1 if it should be made non-solid and 0 if not. This solves the potential problem of many voxels trying to allocate the same brick and requiring a substantial buffer to be allocated in the case of a lot of changes, but it does require that we can set a bit atomically on the GPU, which is challenging.

4 TRANSLUCENCY

4.1 Increasing Voxel Size

The WrldTmpl8 project by default has room for 4 bits per colour channel. Including the emissive component of another 4 bits that is introduced by the ReSTIR project, this means that each voxel requires at most 16 bits. Since we did not want to reduce the range of the other components, we would certainly have to increase the size of each voxel. Given alignment requirements and the setup of the project, the easiest choice was to change the voxels to a type that matched the next largest datatype of 32 bits.

While this would normally give us 16 additional bits of data to work for each voxel, the compression scheme of the grid in its current state assumes voxels of at most 32 bits in size, but uses the last bit as part of the compression itself. Therefore, we have 15 bits left. As described previously, we use four of these 15 bits for an alpha value that describes how opaque an object is and another four have been used to extend the range of the emissive strength.

The change to make voxels larger was straightforward, though several sections of code still assumed 16-bit integers as the voxel type and these were changed subsequently as well to instead use the *PAYLOAD* define.

4.2 ReSTIR and Transparency

ReSTIR as introduced by Bitterley et al. solely concerns itself with direct lighting. Likewise, this is also done by the integration by Hermans in the WlrdTmpl8 project. More precisely, traced voxels will use the reservoir sampling to find the best light to sample for the BRDF, which is the primary strength. This means that the best light will be found for direct lighting, but for additional rays that arise as a result of the BRDF, no such optimal light selection information is possible. This is a problem for rays that end up having to go through surfaces or are reflected by them, as well as any form of global illumination that requires ray bounces. We therefore have to make a compromise: voxels that are translucent are themselves not lit. This allows us to remain using ReSTIR, but rather for the first fully opaque surface behind it.

In order to apply transparency, we need to be able to trace rays from the original hitpoint we found. The ReSTIR implementation uses multiple OpenCL kernels to get the final result, of which two are relevant here: the original albedo gather and the final colouring. The albedo gather is the stage in which the raw colour value is obtained, whereas the final colouring stage uses this alongside the lighting information to obtain the final output. Since the lighting is a multiplication, we can apply transparency both in the albedo gather stage as well as the final colouring stage, as it is an associative and commutative operation.

We chose to do this process in the albedo gather stage, where the initial rays are traced as well. This allows us to use more information without having to store it in memory for subsequent stages and opens up the possibility of combining the two ray traces into something that is more efficient.

Finally, ReSTIR also needs to be able to determine occluders of light in order to correctly retain reservoirs. Normally, any voxel in between the one traced and the light would be occluding, but this is not the same for transparent voxels. We therefore chose that the rays that are shot into the scene to determine occluders will only hit voxels that are fully opaque. In the future one could make it so that the alpha value is used for partial non-binary shadows or to use a better cut-off, for example at an alpha of 0.5.

4.3 Implementation

The idea behind translucent surfaces is that each object, in this situation individual voxels, will have an alpha value associated with them. An alpha value of one will indicate that the voxel is fully opaque, an alpha of zero means we can see fully through it and is practically invisible. As noted previously, we increased the voxel payload size to add this as a property for voxels.

We apply transparency using the commonly found formula in equation 1, where i is the current surface index in the chain of translucent surfaces from front to back, a_s is the alpha value of the surface in front, rgb_s and rgb_d the colour values of the surface in front and behind respectively, and rgb the resulting colour from the blending of the colours.

$$rgb_{si+1} = (1 - a_{si}) * rgb_{di} + a_{si} * rgb_{di} \tag{1}$$

During the raytracing in the albedo gather stage, we first check if the object we hit is an object that is fully opaque. If not, we trace a ray to the next object, starting from the last hit surface point until we hit something fully opaque. This still leaves the following situations unhandled:

(1) The alpha value does not yet scale, despite the fact that transmitted energy should go down as a surface is entered.
(2) The algorithm will never stop if each ray keeps hitting a non-opaque surface.

The first problem is tackled using an existing approach as described by Zhang in Ray Tracing Gems II [1]. The key is that we maintain how much *visibility* remains after a ray passes through a translucent surface. We will refer to this as the *remaining visibility*. An important observation is we can calculate the *remaining visibility* through simple multiplication, which is associative. The associative property allows us to traverse the path from the camera to the light while only maintaining a single 32-bit value for the *remaining visibility* and no need for a stack. We refer to the reference material for more details on the reasoning behind this approach being valid. The primary aspects of our algorithm implementation are sketched in listing 4.

Problem 2, the lack of a stopping condition is still unsolved. Our *remaining visibility* will never be zero as long as we hit an non-opaque surface. We may keep hitting countless successive translucent surfaces and this behaviour would be correct, but at some point such a large percentage of the energy will have been absorbed that the difference will no longer be observable. Continuing at a certain number of bounces would severely impact performances, so we limit the number of rays traced to ten. This limit can quite easily be changed.

Finally, as mentioned in the previous section, we also needed a way to be able to trace rays that would only hit fully opaque voxels. We generalised the macros in *trace.cl*: rather than only skipping over empty voxels, a check can now be passed that is executed on the voxel value and it will only return the voxel as result of the trace if the condition evaluates to true. In practice, this generalisation can be utilised as shown in Listing 5. A function *TraceOpaqueRay* has been added that uses the *GRIDSTEP_OPAQUE* macro, which is then used by occlusion queries.

4.4 Volume Traversal

The approach listed before is simple and generally effective, but could be improved upon. First, we have the problem that we can hit the same voxel twice: once from the outside and once we shoot the second ray that starts inside the voxel, we hit that same voxel again. Second, our origin is moved to the surface point we hit, but we need to add some bias so that it will not hit the same surface at the same point again. This bias can lead to slightly incorrect results, as it can skip over voxels that are grazed at an angle, so ideally we would minimise this effect. Finally, if a ray is shot from every transparent voxel, a brick or larger volume of identical voxels can cause an enormous hit in performance. Additionally, when looking at such a volume at an angle, adjacent pixels will have different numbers of rays to shoot and may therefore stall the warp by having to wait for the pixel that has the longest chain of consecutive transparent voxels. In a raytracer based on triangles, this volume would normally be treated as a single object instead and only use a single ray to traverse through its entirety; the volume of the object is implicit instead.

Listing 4: Alpha blending and calculating the remaining visibility

```
float distance;
float3 origin = ...;
float3 direction = ...;
PAYLOAD voxel = TraceRay(origin, direction, &distance,
    ...);
float remainingVisibility = GetAlpha(voxel);
float3 outputColor = (1.0f - remainingVisibility) *
    ToFloatRGB(voxel);

while (remainingVisibility > 0.f)
{
    origin += distance * direction;
    voxel = TraceRay(origin, direction, &distance, ...);
    outputColor += ToFloatRGB(voxel) *
        remainingVisibility * GetAlpha(voxel);
    remainingVisibility *= (1 - alpha);
}
```

Listing 5: Generalisation of the ray tracing macros in *trace.cl*

```
#define GRIDSTEP_ALL(exitLabel) GRIDSTEP(exitLabel, v !=
    0)

// This traces only fully opaque voxels. The v != 0
    becomes implicit via the alpha check
#define GRIDSTEP_OPAQUE(exitLabel) GRIDSTEP(exitLabel,
    GetAlpha(v) == 0xF)
```

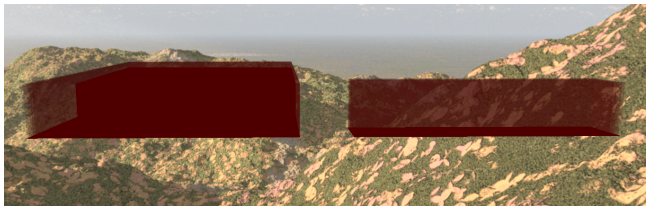

Figure 1: An example of beers law. Both volumes have the same alpha value, but the one on the left is a longer continuous volume from front to back. It is also more darkened then the one on the left, as expected

The last mentioned gives us a look into how we can optimise this approach for the voxel world instead. Listing 5 already shows that we can trace rays while excluding certain voxels. Once a ray hits a translucent voxel, we know that any subsequent voxels that have the same colour and alpha value will not give us any new information, provided we know the distance the ray had to travel to get through the volume. The calculation is the same as in Listing 4, except we use the newly added **TraceRayThrough** instead of **TraceRay**. **TraceRayThrough** does not return for voxels identical to the provided value. A mask can also be provided, as in this way we can perform an equality checking only on the colour values and ignore the emitter strength stored. While emissive translucent voxels are not explicitly supported, this algorithm can deal with it. This approach also eliminates the first problem: we cannot hit the same voxel on its frontside and backside, as they yield identical voxel values and it will therefore trace through the backside. The second mentioned problem is also reduced, as only need to add the bias after every volume rather than every voxel.

Finally there is one problem remaining: no matter the thickness of the translucent volume, the energy absorbed will be the same. For basic games this is not abnormal behaviour, but it is not realistic and does not follow Beer-Lambert's law. This law dictates that the absorbance scales with e^{-da}, where d is the distance through the volume and a the alpha, or in other words, the absorbance rate of the material. Our implementation of this is illustrated by Listing 6. The essential change is that we cannot apply the translucency as we find the ray hitting with the start of the translucent volume. We will need to know the distance it will traverse through the volume that starts at the point where we hit the ray. Therefore, we store the alpha and color for the next iteration rather than applying it to the previous.

We did notice some slight artifacts with this that we did not manage to fully resolve, but the intermediate results can be observed on the branch *experimental-beers-law* in the git repository. An example is shown in Figure 1.

4.5 Skydome

The WlrdTmpl8 project includes the rendering of a skydome around the world. This skydome is implemented in the final colouring stage, where if the ray has not hit any voxels in the scene, the skydome is sampled instead and taken as the final colour. If we instead encounter a translucent material that does not fully absorb all light, the skydome should be visible through the translucent

material. Therefore, we apply the same alpha blending at the end using equation 1 again, but with the *remaining visibility* as a_s.

5 CRAFTING THE DEMO

5.1 Extra bytes for the emissive strength

In the original code base, only 4 bits were used for the emissive strength, which only allowed for a range of values between 0 to 15. This was not enough for larger scenes. Initially we experimented with a global multiplier for the emissive strength, but soon enough we decided to use the remaining bytes after the 32bit extensions to bring the brightest value to 255.
The final payload layout is then the following: **0x00MMARGB** where:

- **M** is the emissive strength (8 bits);
- **A** is the alpha value (4 bits);
- **R** is the red channel (4 bits);
- **G** is the green channel (4 bits);
- **B** is the blue channel (4 bits);

5.2 The demo

The demo shipped with the project is composed by a lighthouse on a small island. The lighthouse and the island have been modelled using Magicavoxel and imported as a sprite.
The lighthouse sprite itself has 5 different animation frames that can be toggled with the spacebar to rotate the portion that covers the light. This will result in some portions of the it to be hidden by solid voxels, hence the illumination on the scene will change.
The rotating part is separated by the terrain and the other elements of the lighthouse so that those can be stamped into the world with **StampSpriteTo**, making those voxels editable trough the world editor.
Some extra lights in the form of buoys are added around the scene, along a small boat with a lantern. The overall result, with the added semi-translucent water, can be seen in Figure 4.

Listing 6: Applying Beer's law for correct absorbance

```
float distance;
float3 origin = ...;
float3 direction = ...;
PAYLOAD voxel = TraceRay(origin, direction, &distance,
    ...);
float remainingVisibility = GetAlpha(voxel);
float3 outputColor = (1.0f - remainingVisibility) *
    ToFloatRGB(voxel);

while (remainingVisibility > 0.f)
{
    origin += distance * direction;
    voxel = TraceRay(origin, direction, &distance, ...);
    alpha = 1.0f - exp(fabs(-dist) * alpha);

    outputColor += ToFloatRGB(voxel) *
        remainingVisibility * GetAlpha(voxel);
    remainingVisibility *= (1 - alpha);
    alpha = GetAlpha(voxel);
    color = ToFloatRGB(voxel);
}
```

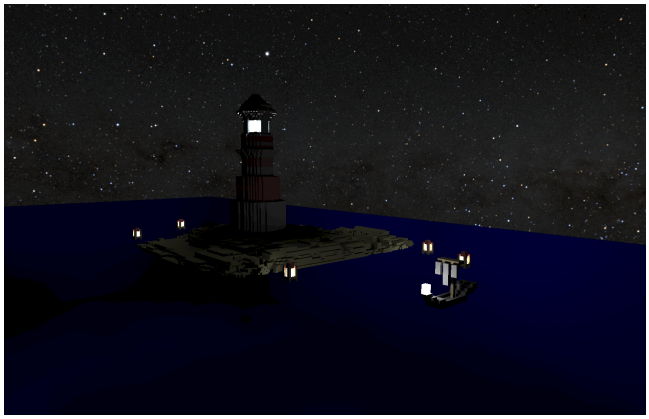

Figure 2: Screenshot from the demo scene

5.3 Performance

We tested the performance of the project both with and without translucent voxels in the demo scene by taking the average frametime of the *albedo gather* stage of the rendering. We decided to only the frametime for this step, as making the voxels translucent changed the scene significantly enough to also affect the frametime for the other stages as well.
Without translucent voxels, the average frametime taken with a rolling average over a 100 frames was approximately *0.39ms*. With translucent voxels, the average was *1.08ms* instead. To put this into context, the total render time without translucent voxels was approximately *53.84ms*. Adding translucent voxels slows down the rendering time as expected, but it still accounts only 2% of the total render time.

6 WORLD EDITOR

This section details the varying functionalities of the world editor, its limitations, and suggested future work. The **WorldEditor** class was created and added to the template. The WorldEditor can be enabled by pressing the G key and is disabled by default. All of the relevant source code can be found in `WorldEditor.cpp`. For this section the term object will refer to anything that can be placed by the editor i.e. Sprite, Voxel, Brick, Tile, and Big Tile.

6.1 Initialisation

The editor is initialised after the world. This is because the world editor initialisation relies on usage of the world's various buffers. The editor also keeps copies of the world grid, brick buffer, and zeroes buffer. After these copies are initialised, we load all of the assets that can be used by the editor. The editor supports three types of assets:

(1) **Tiles** (.vox files of size 8x8)
(2) **Big Tiles** (.vox files of size 16x16)
(3) **Sprites** (.vox files of any size)

These assets are required to be placed within the assets/(asset name) directory within the project in order to be loaded into the editor. Upon loading an asset, the editor checks to see if there is a .png file with the same name to use as a preview. If one does

not exist, the editor will add the asset to the world at the origin, and raytrace a preview image by determining a suitable camera position dependent on the size of the asset and a fixed angle. This raytraced image is then written to a 256x256 .png with the same name as the asset. The lighting uses the default SkyDome lighting when the world is created so it's important this step occurs before the Game class overrides any of these values. This approach to generating previews works for most cases, however for certain vox files it is possible for the preview to "cutoff" the asset. See **WorldEditor::LoadAssets** for full implementation.

6.2 Rendering and GUI

When the editor is enabled a grid is displayed that surrounds the world in order to clearly see the world boundaries. This intersection uses a ray box intersection from [3]. Special cases are made for when the ray origin is outside of the grid so only the second hit grid wall is displayed. This removal of sides of the grid allows us to look into the world from all angles when outside of the grid.

It is beneficial for the user to visualise where they will be placing objects. To do this, we render a wire frame box the size of the desired asset at the point selected by the mouse. This is achieved with the **Selected** struct. This struct contains an AABB to define the bounds of the box as well as an "anchor" position which is used when changing the box dimensions during a gesture. We trace a ray from the camera origin to the point selected by the cursor. If we hit the grid (the edges of the world), then our box is placed at the nearest point to the intersection while ensuring the box does not extend past the world boundaries. If our mouse intersects with an object, we compute the intersection normal and place the box at the next free position in the direction of the normal. The box is rendered by sending the bounding box to the GPU and detecting when a ray intersects with the box edges. The size of the wire frame depends on the type of object being added. See **WorldEditor::UpdateSelectedBox** and **HitSelectedBox** in tools.cl for full implementation.

For the GUI we use DearImgui to create four key interfaces:

(1) Main Menu
(2) Camera Details
(3) Asset Hot Bar
(4) All Assets Preview

6.2.1 Main Menu. The File menu allows us to clear the world, save the world, or load an existing world. The Edit menu allows us to Undo and Redo editor gestures as well as optimize any bricks. The View menu gives the user a choice of what windows they want to display as well as disabling or enabling rendering of the grid. We also display render times of the albedo stage as well as the total render time.

6.2.2 Camera Details. This window displays the camera position and view direction as three floats. These values can be edited within the window and require the enter key to be pressed for the camera to use these new values.

6.2.3 Asset Hot Bar. The hot bar enables switching between editing with the three different asset types as well as with Bricks/Voxels. For the assets, up to eight previews are generated and displayed in the hot bar. Clicking on the preview will add a red outline and update the editor to use that asset when editing. These previews can be rearranged by dragging and dropping the preview into the desired order. When using Bricks/Voxels, the user can click on the color to open the color picker and set the desired color and alpha to be used when editing. The alpha bar is used to determine the materials transparency. There is a horizontal slider next to the color picker to set the desired emissive value. Additionally, there is a checkbox which switches between editing with Bricks or with Voxels.

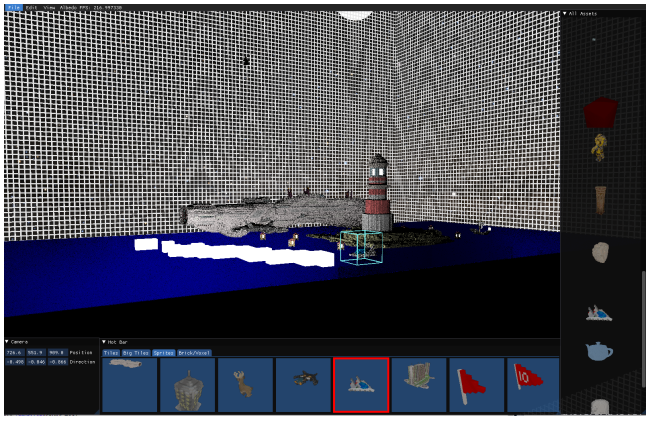

Figure 3: Screenshot of the world editor in use with the world grid being rendered

6.2.4 All Assets Preview. This window displays previews of all of the loaded assets. In order to use these assets, the user must drag and drop the preview into the associated hot bar tab. Then the user must select the preview within the hot bar. The previews within this window cannot be rearranged and are displayed in the order they are loaded into the editor.

6.3 Gestures

We use the term gesture to define an action the user is taking with the editor. At the moment the user is able to **Add**, **Remove**, **MultiAdd**, and **MultiRemove**. When the user clicks to begin a gesture, the state of the world (grid, bricks, zeroes) is copied into the associated buffers. This is because if we are adding multiple objects at once by holding the mouse down and dragging, we want to trace the world in the state it was in before any new objects were placed.

There is a **Gesture** struct that stores what keys are pressed, what mouse button is pressed, whether or not we can perform another gesture, the type of gesture to perform, and the size of the gesture (1x1x1 for voxels, 8x8x8 for tiles/bricks, etc.).

6.3.1 Add. A user can add an object by simply clicking the left mouse button. This will then place whatever object is selected in the Hot Bar window at the position displayed by the wire frame box. If the user holds down the left mouse button and moves the mouse then multiple copies of the object will be placed. The **Add** method takes in voxel coordinates and will convert them to relevant coordinates for Bricks, Tiles, and Big Tiles. Bricks, Tiles, and Big Tiles can only be placed in specific locations within the world. These

positions must be multiples of 8 for Bricks and Tiles, or multiples of 16 for Big Tiles.

When placing a Sprite, the sprite is stamped into the world meaning it will not have the behaviour of a sprite.

6.3.2 Remove. Depending on the size of the selected object in the hot bar, the wire frame box will vary in size. When the user holds down the shift key and then presses the left mouse button, everything within the box will be removed from the world. The **Remove** method takes in voxel coordinates and will convert them to relevant coordinates for Bricks, Tiles, and Big Tiles.

6.3.3 MultiAdd and MultiRemove. If the user holds Ctrl and drags the mouse while adding or removing, the wire frame box will expand in size with one corner at the intersection of the initial click and the other corner at the intersection of the current mouse position. This will add/remove multiple objects simultaneously. As long as the mouse button is held down, the box can be resized and objects will be added/removed appropriately. Note that Sprites cannot be used with these gestures at the moment.

When the selected box is updated in size, we don't want to add/remove objects that have already been updated in the previous box. To achieve this, we calculate the intersection between the old box and the new box. Anything changes that happened in the old box but not the new box are reverted. No changes are applied to objects within the intersection. Any part of the new box that was not part of the previous now applied the desired changes.

This was the most difficult gesture to get working as it requires constant updates to the grid and brick buffer. In addition, the trash buffer needs to be kept in sync. This was achieved by removing and allocating bricks on each update instead of purely copying grid and brick values from one buffer to another.

6.4 Editor States

Once a gesture has been completed, a **state** object is created which stores relevant info about what was updated during that gesture. This information includes old and new grid values, the number of bricks that were updated, the old and new values of updated bricks, etc. Once the state is created, it joins existing states as part of a history. Each state has a pointer to the state before and the state after (if one exists). This allows us to Undo and Redo gestures either by using the main menu or by pressing Ctrl+Z for Undo and Ctrl+Y for Redo. As we need to allocate memory to store the state information, the `MAX_ALLOCATION` threshold macro is checked whenever a new state is added. If the total memory used by all states exceeds this threshold then we begin to delete the oldest states until we go under the threshold again.

6.5 Saving and Loading

It is possible to save and load worlds into the editor. The file name used when saving/loading is worlddata.nbt and the corresponding format can be seen in Listing 7. This file is saved and loaded from the project directory. The nbt file format was selected as this is the format used by Minecraft and it may be the case that future work includes the ability to load data generated from Minecraft. There are a series of helper functions under the namespace `NBTHelper` to assist in the writing and reading of these files. When saving the

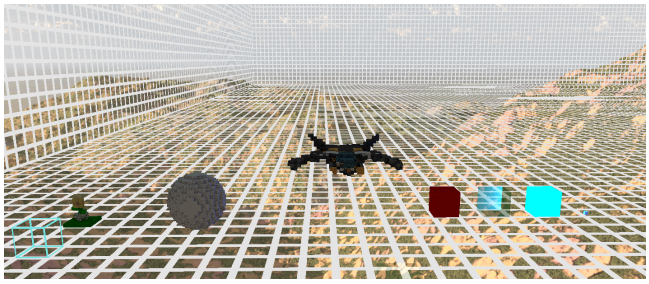

Figure 4: Screenshot of the world editor after loading the supplied worlddata.nbt file

world, we write out a list of grid values and indices. We also write out a list of brick buffer offsets and payload values. When loading, we clear the world completely resetting the entire state and then add all of the bricks and grid values into the correct buffers. When adding in bricks, we call `NewBrick` to ensure the trash buffer is using the correct indices. Afterwards, we setup the light buffer and remove all previously saved states to start fresh. It is not possible to Undo or Redo the loading of a world. An example worlddata.nbt file is included in the repo which loads all different objects.

Listing 7: NBT File Format

```
NBT File format
Tag_List("grid list") : x entries of type
     Tag_Compound (Note: Elements in list are unnamed
     )
{
  TAG_Compound : 2 entries
  {
    TAG_Int("grid index") : index
    TAG_Int("grid value") : value
    TAG_End
  }
  ....
}
 Tag_List("brick list") : x entries of type
      Tag_Compound
{
  TAG_Compound : 2 entries
  {
    TAG_Int(""brick buffer offset"") : offset
    TAG_Byte_Array("brick value") : [PAYLOADSIZE *
         BRICKSIZE bytes]
      TAG_Int("brick zeroes") : zeroes
    TAG_End
  }
  ....
}
```

6.6 Limitations and Known Issues

As previously mentioned one limitation is the inconsistency to generate a preview for all different Sprites. Another limitation is that when adding the same tile/brick we aren't reusing existing bricks within the brick buffer and are instead adding copies of the bricks. A quick profiling of the editor shows that adding lights is the most time consuming operation. MultiAdding lights results in the light buffer being resized frequently and copied to the GPU every time an individual light is added. This could be resolved by waiting for all lights to be added to the world before updating the

lights buffer, or by waiting until the gesture is completed before proceeding.

There is currently an issue with the number of lights not being appropriately set when MultiAdding an object over an existing light and then moving the cursor so the change is undone. The number of lights in the light buffer increases when it should remain the same as before the MultiAdd gesture started. There is also an issue when editing while sprites are being drawn. When adding lights to a position that is being covered by a sprite and then removing the added light, there remains lighting artifacts on the rendered sprite for some unknown reason. Unfortunately, there was not enough time to fully investigate these aforementioned issues.

6.7 Future Work

With an editor such as this, there are many additions that can be made. We've compiled a brief list of ones we thing would benefit the editor the most:

- Adding the ability to create Tiles/Big Tiles/Sprites within the editor
- Being able to load new vox models from the editor instead of relying on the file directory
- When placing the same Tile/Big tile multiple times, keep a reference to the relevant bricks within the buffer instead of creating new bricks.
- When selecting a tab on the hot bar the same tab should be chosen on the all asset windows and vice versa.
- Being able to place Sprites in the world instead of just stamping them. This should be straightforward with the move methods associated with sprites.
- Add a rendering window to allow the user to change render parameters from the GUI instead of through code/key presses which are inconsistent across games
- Add a window to allow the user to change the gesture method from GUI
- Loading and saving worlds using unique filenames

REFERENCES

[1] Adam Marrs, Peter Shirley, and Ingo Wald (Ed.). 2021. *Ray Tracing Gems II.* Apress, Chapter 11. http://raytracinggems.com/rtg2.

[2] Xander Hermans. 2022. *The effectiveness of the ReSTIR technique when ray tracing a voxel world.* Master's thesis.

[3] Alexander Majercik, Cyril Crassin, Peter Shirley, and Morgan McGuire. 2018. A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering. *Journal of Computer Graphics Techniques (JCGT)* 7, 3 (20 September 2018), 66–81. http://jcgt.org/published/0007/03/04/

[4] William Van der Scheer. 2022. *Cellular Automata Fluid Physics for Voxel Engines.* Master's thesis.