COMP 3501 - Game Development Project
Brady Jessup - *101001590*
Forest Anderson - *101009514*
Lachlan Campbell - *100999056*

Operation Skullrain

Final Report

Table Of Contents:

COMPILATION NOTE: Operation Skullrain must be run in RELEASE mode. DEBUG mode will not work properly. You may also have to retarget the solution. In visual studio go to Project > Retarget Solution > Select the Windows SDK version you are running (should be the only one available) > Press OK.

Game Overview:

You are a rescue helicopter tasked with protecting the city of Skullrain. A massive forest fire has broken out around the city, spelling certain devastation if you don't stop it from reaching the city. Your task is simple; put out the forest fire all the while dodging incoming flaming projectiles and making sure you have enough resources to do so. You also must try and rescue as many civilians outside of the city as possible and bring them back to the city where it is safe (for now).

The gameplay itself is fairly lacking as some of the functionality wasn't able to be implemented. However, the engine makes up for it with its large array of features such as, an integrated skybox, 3D multi-textured terrain, a hierarchical scene-graph for helicopter animations. The engine also boasts anti-aliasing and anisotropic filtering. All this and much more.

Game Manual:

- Controls:
    - Space: Fly Up
    - WASD: Forward (W), Backward (S), Left (A), Right (D)
    - L_Control: Fly Down
    - 1: Shoot
    - 2: Use Shield
    - C & V: Toggle third (V) /first (C) person camera
- Objectives:
    - Destroy the towers hurling projectiles at you
    - Put out the flames
    - Rescue as many civilians as you can

Technical Requirements:

Requirement	**Method of Integration**
General Fire (Enemy) [Figure 2.3]	Only implemented for the player death animation. The Flames will spread upon the player's death from the enemies (or you can hold k to kill yourself in-game)
Projectiles (Enemy) [Figure 2.2]	Projectiles will fire from randomly spawning towers around the terrain. The projectiles will damage the play if they don't either dodge them, blow them up or have their invincible shield on.
NPC's (Obstacle) [Figure 2.1]	NPC's will randomly spawn around the map. They will walk toward your helicopter when they are within a certain radius around the helicopter. When they get close to the helicopter they will go inside of the helicopter and you can view them in first - person mode.
Water Gun (Weapon) [Figure 1.2]	The player can press 1 for the primary weapon fire. It has a 5 second cooldown after it is used, and will fire for 2 seconds. It is designed to be used against towers and other projectiles.

Invincibility Bubble (Weapon) [Figure 1.3]	The invincibility bubble appears as a 'skin' surrounding the helicopter using the renderer's light refraction. No projectiles can hit the heli whilst active.
Free-flying Helicopter [Figure 1.1]	Helicopter flight programmed on 3-axis. Has the feel of a real-life helicopter as accelerating tilts the heli forwards and vice - versa for decelerating.
Third & First Person Camera Support [Figure 1.1] [Figure 3.6]	Being able to switch seamlessly between third and first person view is achieved by a boolean value on the camera object. If third person is true, the camera position simply shifts outside of the helicopter.
Collision detection between helicopter and other objects (NPC's, enemies, terrain, projectiles etc)	The helicopter currently collides with the terrain (which isn't flat but instead it uses a heightmap). It also collides with NPCs and they will even approach you once you get close enough.
Hierarchical rotation of the rotor blades and turrets on helicopter	Using quaternions and a local hierarchical scene-graph adaption.
Lighting [Figure 3.1 - 3.3]	Blinn-Phong lighting algorithm. With support for point lights, spotlights and directional lights. (all showcased within the game)
Cubemap Support [Figure 3.4]	Custom shader to render skybox with.

Additional Features Added:

- Support for loading models and their materials (diffuse maps, specular maps)
- Framebuffer creation with post processing capabilities (not in use, but is there if needed, we have the shaders and render pass already occurring, we just aren't applying a convolution matrix) Our framebuffer also supports MSAA and will blit the multisampled framebuffer to a non-multisampled buffer so we can perform post processing [Figure 4.2 - 4.3]
- Terrain rendering that supports multi-texturing (up to 5 textures: using a blend-map). And uses a heightmap to achieve proper hills and valleys. [Figure 4.1]
- Basic Blending support, and the renderer will handle transparent objects by disabling back face culling and rendering back to front. [Figure 4.4]
- Fullscreen support
- Shaders for reflection and refraction using our skybox cubemap (basic environment map) [Figure 3.5]
- MSAA support (Multi-sample anti-aliasing) [Figure 4.5]
- Anisotropic filtering support, this greatly increases the quality of the terrain [Figure 4.5]
- Simple 2D UI [Figure 2.2]

Features Not Implemented:

- General flame algorithm
    - Burning trees
    - Spreading fire algorithm
- City drop point
    - City Model
- Refueling
    - Water refueling
    - Fuel refueling
- Weapon collision detection
    - Between weapon projectiles and enemy projectiles
    - Between weapon projectiles and towers

Technical Notes:

Currently our engine uses a logger to report data in case of a crash or any errors. The logger uses a Singleton design pattern and that gets rid of the conversion and copy constructors, it also accounts for move semantics which were introduced in C++ 11.

Our scene is mostly handled by Scene.cpp. It has instances of the terrain, player, lists of entities to render and update, list of renderables (ie stuff that doesn't update), and the camera. It also has access to the most important thing in the system, the renderer. The renderer uses the model class to set up the textures and bind the corresponding VAO. However it handles all of the other complicated functions like setting uniform input variables in the shaders. It will also modify the stencil and depth buffer parameters in certain situations. It also handles backface culling (ie it is disabled when rendering transparent but enabled for opaque). Finally it sets up the model matrix for rendering, and it will do it in a hierarchical manner so we can have effects like the rotation of the rotors. The camera class will handle stuff like the first and third person camera, it also keeps track of the yaw, and pitch of the camera (no roll). The camera is constrained to certain pitch values but it can have any yaw value.

We also have a renderer that is unique for rendering particles (that occur when the player dies). It sets up stuff that is specific to particles and other OpenGL state that make them look better.

There is also a UI renderer which will render 2D shapes and texture them onto the screen. It also supports size changes that we use for stuff like the player health bar. We would have liked to add more UI to the game, because we spent time on the rendering systems for it, but in the end we didn't have time to add all of our weapons so we decided not to add UI for the one type of weapon the player possesses. The UI renderer will make use of the depth buffer to ensure that the UI is rendered in front of everything else. We also can easily make the UI render first so that there is no overdraw. However we didn't have enough time to implement this properly because of a couple of reasons. For example to implement this properly we would have used the stencil buffer to render the shape of the UI with a certain value. Then when we rendered our scene we would draw the scene normally unless the stencil buffer had those values stored at that location. Then we would perform our post processing pass, and finally we would actually render the UI. This way the UI is rendered properly and isn't involved in the post processing stage, and there would be no overdraw.

We have lots of stuff in our game world like towers, trees, and NPCs. They all need to be generated and placed in the gameworld. To do this we have spawner classes which all inherit from the Spawner class. These classes generate the entities and the renderables for the objects in the game world. However it takes into account the height of the terrain, its position, and its rotation. So we need to define these world space positions in such a way that they respect the terrain space positions. For this we use terrain space when generating and we convert the positions to world space before passing them to the scene for rendering and updating. If we had more time we would have liked to use instanced rendering to greatly increase performance of rendering 500 trees and 25 NPCs etc. On some integrated chips, having 600 draw calls per frame is quite taxing, however we already spent most of our time making the game look pretty.

The scene also handles the skybox, and it is very easy because we have utility classes for certain OpenGL functions like generating cubemaps. I will explain the process of rendering the skybox: We define the position of the skybox around the camera (0, 0, 0), the skybox has dimensions of 1x1x1. Then in the vertex shader we clip off the translation part of the view matrix so the skybox will rotate with the camera and scale, but it will not translate. This is important because we want the skybox to always be around the player. Then we perform a little depth trick in the vertex shader to make the skybox be rendered at the maximum value of 1.0 in the depth buffer. This can be achieved on the following line: gl_Position = pos.xyww
This makes it so when perspective division is performed, w/w = 1.0. So when it becomes a normalized device coordinate it will have a maximum degree value of 1.0. However there are a few more tricks we use to maximize efficiency. Since the skybox will be rendered for every pixel, we don’t want to render the skybox first, since they pixels will become occluded therefore overdraw is occurring. So instead we render the skybox near the end, but we need to render it before the transparent objects so the skybox can be seen through them. However before we render we need to tell OpenGL to pass the depth test if the value is Less than or Equal. Because before when it was just set to less than, the depth test was failing for the skybox since we assume the values in the depth buffer are at a default value of 1.0, because that would make sense. So the rendering of the skybox will take this into account and it will change the function when rendering the skybox but it will change it back after the draw call.

We make use of factory design patterns for constructing particles and basic meshes. This way all of the complicated setup is handled by a class, and we only have to ask the factory to instantiate the object and it will return a pointer to the dynamic memory.

NOTE: As you can see we put a lot of consideration into our rendering. Making use of basic environment mapping when rendering the player, a death animation that uses the normals to explode the helicopter, anisotropic filtering and MSAA to provide a clear image, and many other effects. For these reasons we didn’t get to implement as much of the game as we hoped, but we hope you enjoy what there is and that the visuals make it worthwhile.

Classes taken from the internet as a reference to use ASSIMP to load models:

- Mesh.h Mesh.cpp
- Model.h Model.cpp

https://learnopengl.com/#!Model-Loading/Model

Appendix (Screenshots):

Figure 1.1: Helicopter model

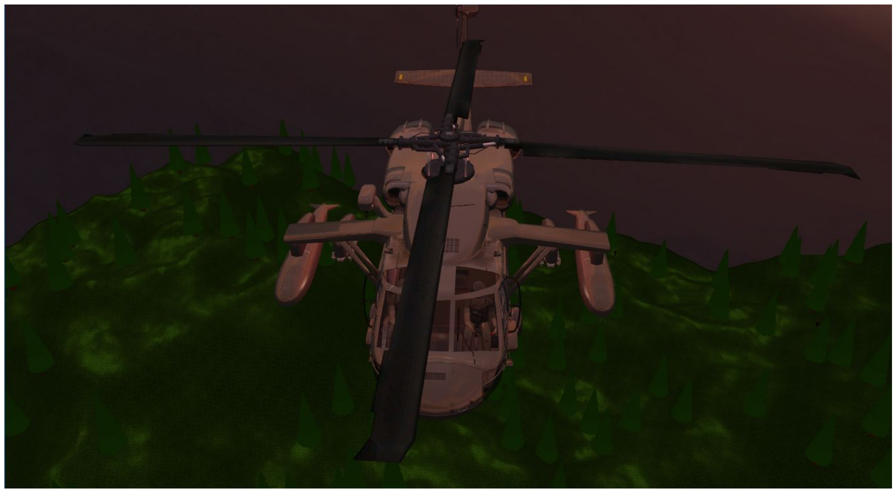

Figure 1.2: Helicopter's weapon (watergun)

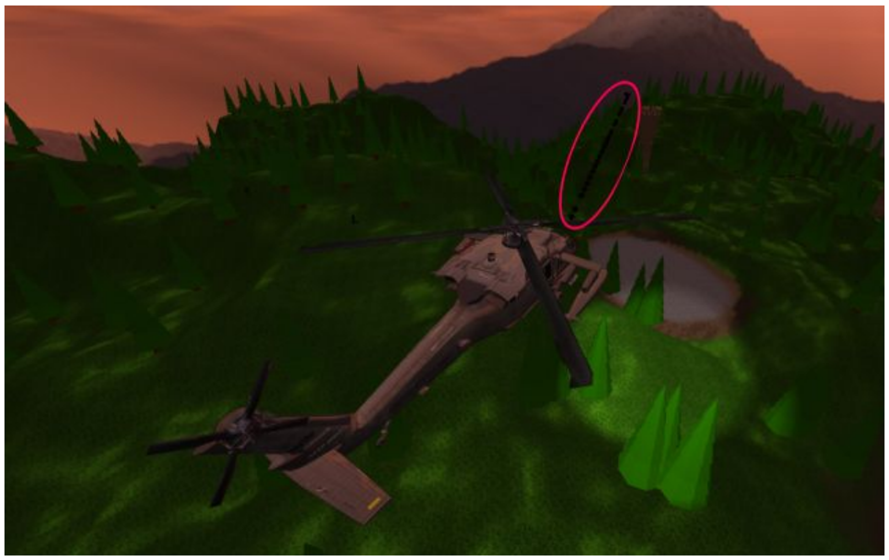

Figure 1.3: Helicopter's Invincibility weapon (uses refraction)

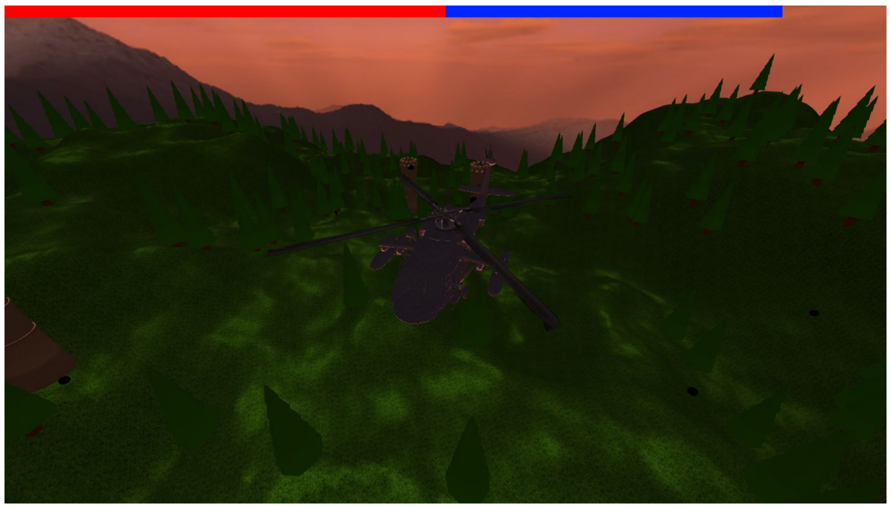

Figure 2.1: NPC’s

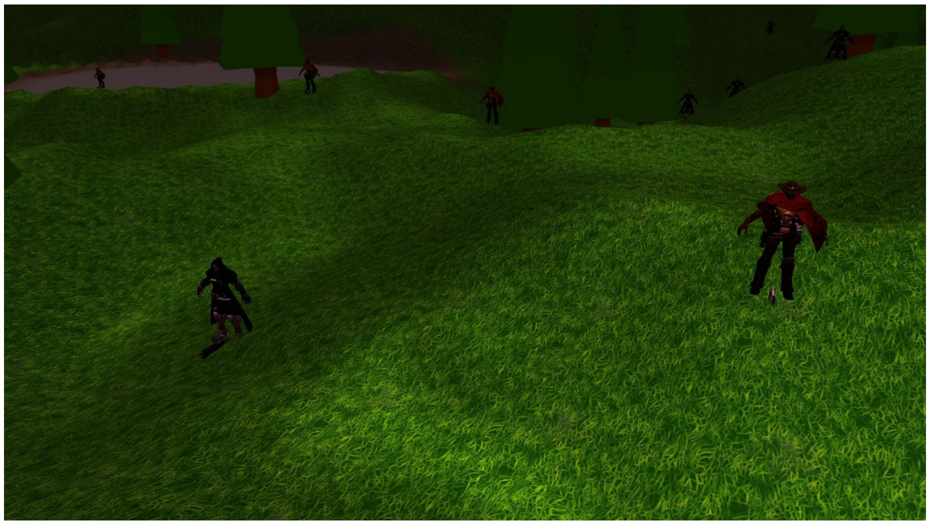

Figure 2.2: Projectiles & UI

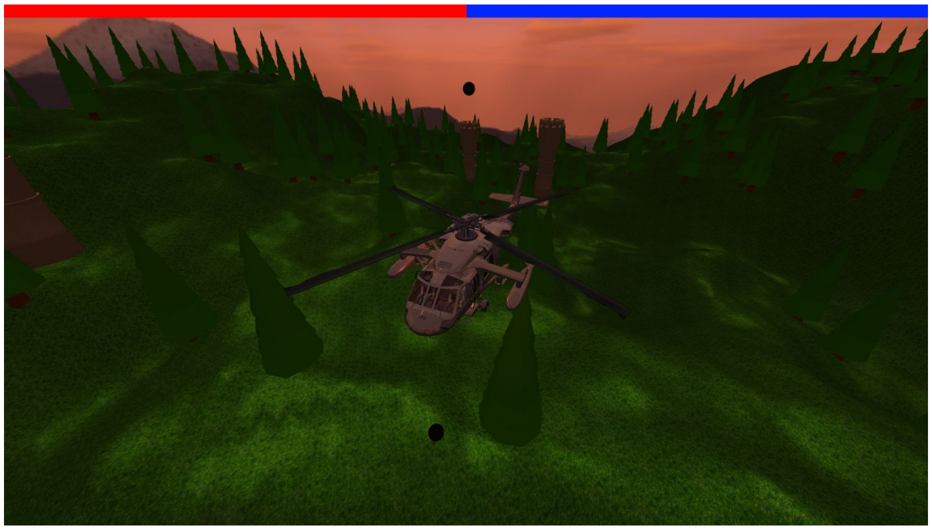

Figure 2.3: General Fire

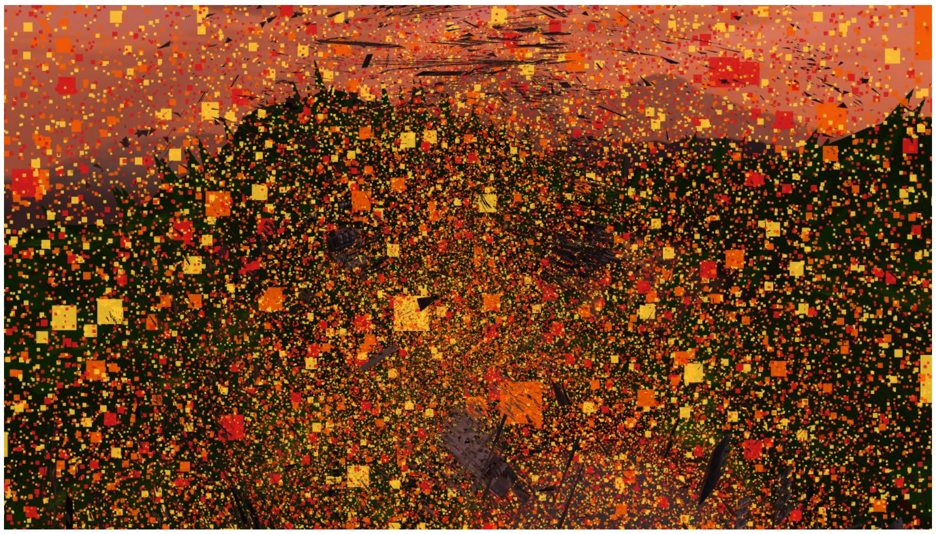

Figure 3.1: Point Light

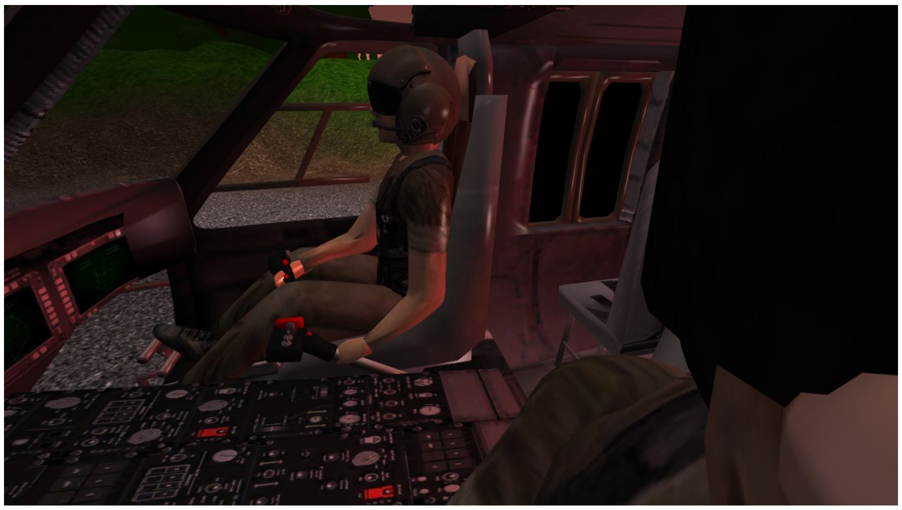

Figure 3.2: Spotlight

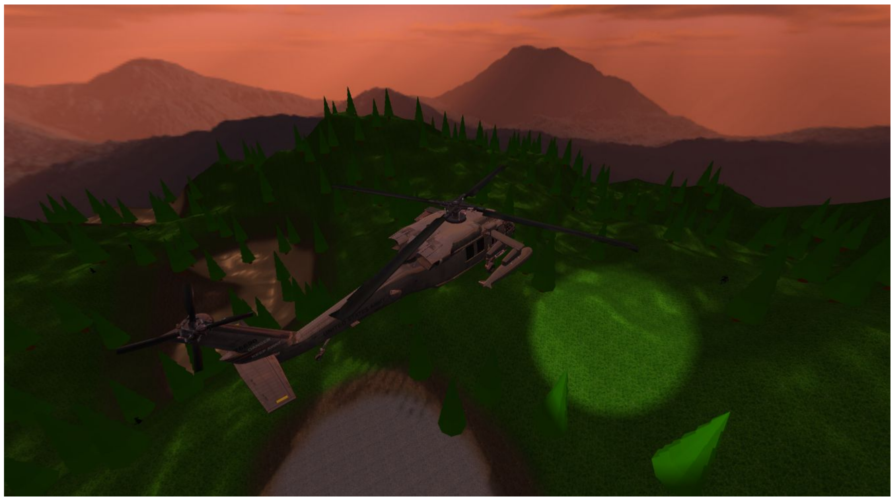

Figure 3.3: Directional Light

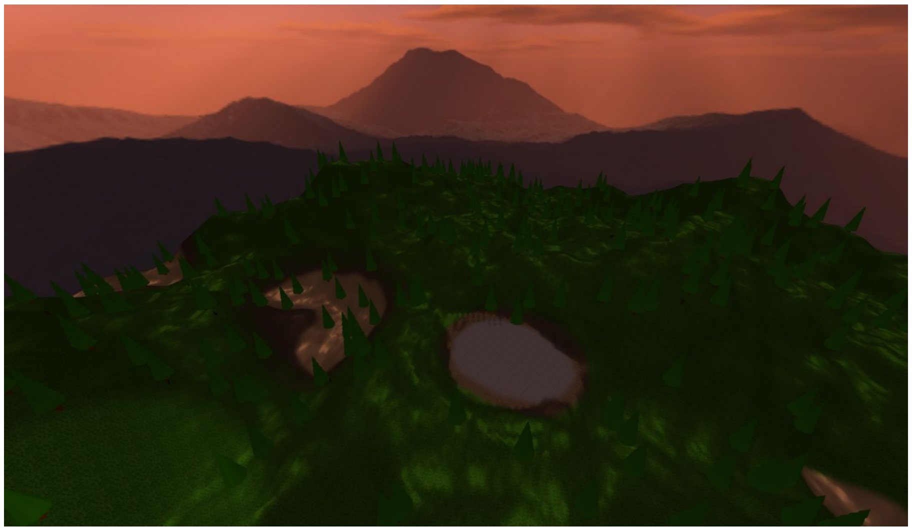

Figure 3.4: Skybox

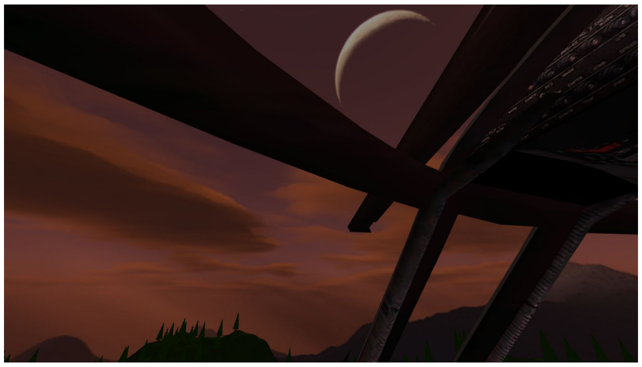

Figure 3.5: Skybox reflection(using environment map)

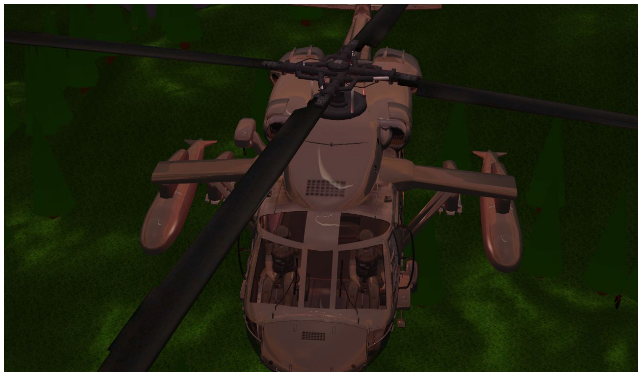

Figure 3.6: First Person View

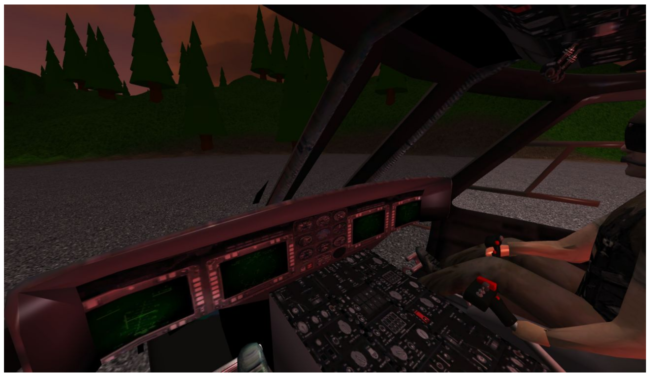

Figure 4.1: Terrain Blending Textures

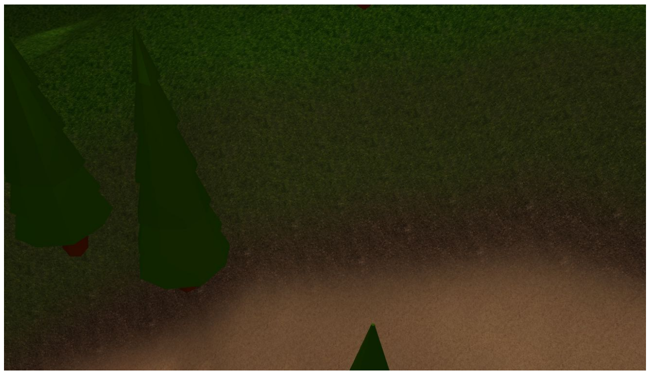

Figure 4.2: Post Processing effect (blur)

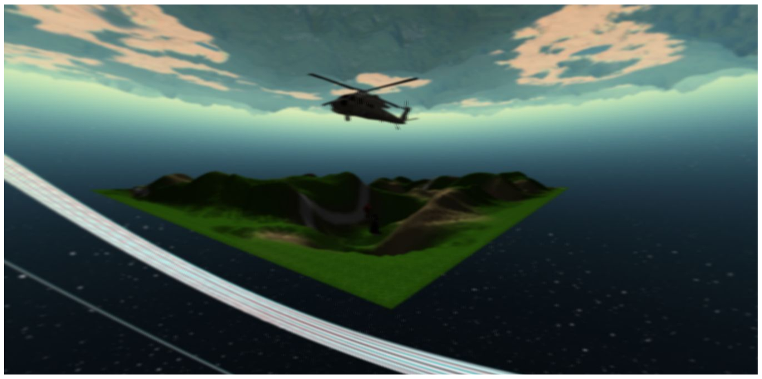

Figure 4.3: Post Processing effect (edge detection)

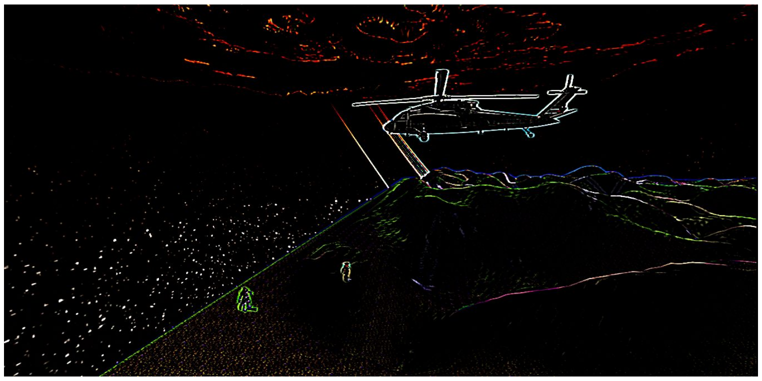

Figure 4.4: Blending Support (back-to-front rendering for transparent objects)

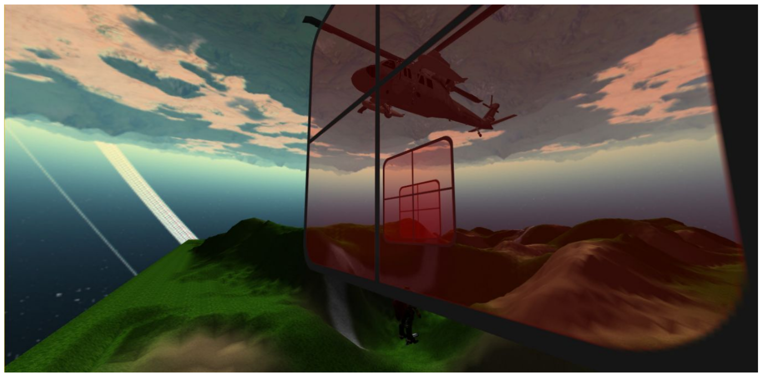

Figure 4.5: anisotropic filtering and MSAA (and our model shading)