Concordia University
Comp 371 - Computer Graphic
Professor: Nicolas Bergeron

Duck Hunt

By Team U

Amrou Abdelhadi ID: 40045799

Ghislain Clermont ID: 40057664

Maryam Eskandari ID: 40065716

Poon Kai Alan Fok ID: 26816797

Robert Laviolette ID: 27646666

Table of Contents

1) Project Overview

1.a) Summary

Team U's Duck Hunt is a three-dimensional reimagination of Nintendo's 1984 game for the NES with the same name. While Team U's Duck Hunt doesn't keep all of the mechanics that make Duck Hunt a game, it captures a lot of the essence of what that game is: there is a duck that flies around the screen, and the player has an in-game gun (representing Nintendo's Zapper gun controller) that they can use in order to shoot the duck. The object of the game is to do exactly that.

More specifically, Team U's version Duck Hunt has a single duck that flies in a circle in front of the player on the screen. The goal of the player is to shoot this duck, although the duck won't die unless duck death is enabled (by pressing [X]). If duck death is enabled, shooting the duck will cause it to die in a similar manner to the NES Duck Hunt (it freezes for a moment, and then falls to the ground and despawns). The player then wins, or they can continue playing by respawning the duck with the [N] key.

Team U's Duck Hunt is made using both C++ and OpenGL, with models created in Maya.

1.b) Expected vs. Actual

Similar to nearly all games, the final result had less features than originally planned. Due to time constraints and the difficulty of some of the other features, certain things were cut from the final product. Among these are a score, UI, shadowing, game logic (such as automatically respawning ducks), and so on. These missing features will be discussed as "Future Work" in the Discussion section.

However, the majority of the game's features managed to make it into the game, as well as a few additional features that were decided on partway through the project. All of these successfully implemented features will be discussed in further detail in the Methodology & Results section.

2) Methodology & Results

2.a) Duck Animation

Methodology

Animations were implemented with the goal of using no extra library than what was already used in the Assignment framework. To keep it simple, the animation and rigging were designed so that bones only extend toward the x-axis of the model, and rotate only around the z-axis of the model. This ended up being a life-saving restriction.

The duck was given 2 bones per wings, and one "main" bone that also controlled the translation on the y-axis of the model. The weights were painted directly in Autodesk Maya, and then extracted using the

“Extract Weight Maps” feature, which creates an image file for each bone based on the UV, and colored on a scale of black to white. The animation was also done directly in Maya and then exported into a *.atom* file. Although this is a human-readable format, the format was complicated and contained a lot of useless information. As such, the values were copy-pasted into a customized format inside the *.scene* file.

The animation and weights were then imported into the project, saved into the new classes *BoneAnimation* and *AnimatedObjModel*, respectively. *BoneAnimation* takes care of calculating the position of each bone on every frame, and apply these transformations onto a given vertex. *AnimatedObjModel* takes care of saving the original position of the bones as well as the weights, builds the new *Model* by passing every vertex into the attached *BoneAnimation*, and updates the vertex buffer that is sent to the GPU. The model can then be attached both a path animation and a bone animation, making it possible to choose how it moves in the world without changing animation.

Result

Due to each color in an image file being an integer between 0 and 255, this caused imprecisions into the imported weights. This caused the back of the wing to sometimes clip into the front of the wing at the extreme of the wing flapping animation, an issue that is common with thin, two-sided parts of models.

Performance-wise, the first attempt was very horrible. Despite having only four animated bones and 2214 faces, it dropped the frame rate to approximately 4 FPS. The following optimizations were required to finally reach a presentable 30 FPS:

- The calculation of each bone rotation based on the current frame of their animation is done once at the beginning of each frame, instead of once per vertex.
- The Matrix that results from a single bone transformation was precalculated based on the limitations of the animation. By passing the position of the bone and its current angle, the program assigns directly each value of the matrix, dividing by 3 the amount of matrix multiplications required.
- Since most of the time, the same vertex was reused in the previous face, the program checks in the last 3 calculated vertices to avoid useless calculations.
- The transformation of the normals was disabled. This caused the lighting to not properly update on the duck’s wings.

- The amount of faces on the duck was cut by 60%, reducing to 886 faces. This amplified the problem of the wing clipping into itself.

The animation itself however was very good. The motion of the wing animation was identical to what was designed on Maya, the duck could easily change animation by passing a new *BoneAnimation* onto it, and the way it was coded allowed us to easily "freeze" the duck's animation for when it was shot, keeping a reference to the original Duck Hunt where the duck does not fall immediately after being shot.

2.b) Models

Methodology

A total of four different models were created: The ground, the trees, the gun and the duck. All of these models were created in Autodesk Maya and exported into OBJ files. The UVs of the models were also unwrapped in Maya. A snapshot of the UV was then exported into Photoshop to create the Texture files according to the position of the UVs. Finally, every OBJ was imported back into the project using the OBJ loader from the labs, with a few changes to fit the assignment framework. The class responsible for doing this is the *ObjModel* class, which inherits from the *Model* class, reads the *.OBJ* file in the constructor, then saves the data to be used in the same way as the *SphereModel* and *CubeModel* classes. Since OBJ files don't have any color information, no color is sent to the shader.

Every model had a few peculiarities:

- Ground: The ground was designed from the perspective of the camera's location. The ground where the player is at is flat and located at 0 on the y-axis. Hills also prevent the player from seeing where the ground model ends, while also giving locations where the duck could appear without looking like it just clips out of the ground. This last feature was unfortunately not implemented.
- Trees: The same model was used for all 3 trees. However, a rotation around the y-axis, and a slight change in scale was given to make them look different.
- Gun: Since the gun must follow the camera, a quick hack was done where the world matrix is multiplied by the inverse of the view matrix if the name of the object is "gun".
- Duck: As the duck needed an animation, it used a different class, *AnimatedObjClass*. See the Animation section above for more details.

Results

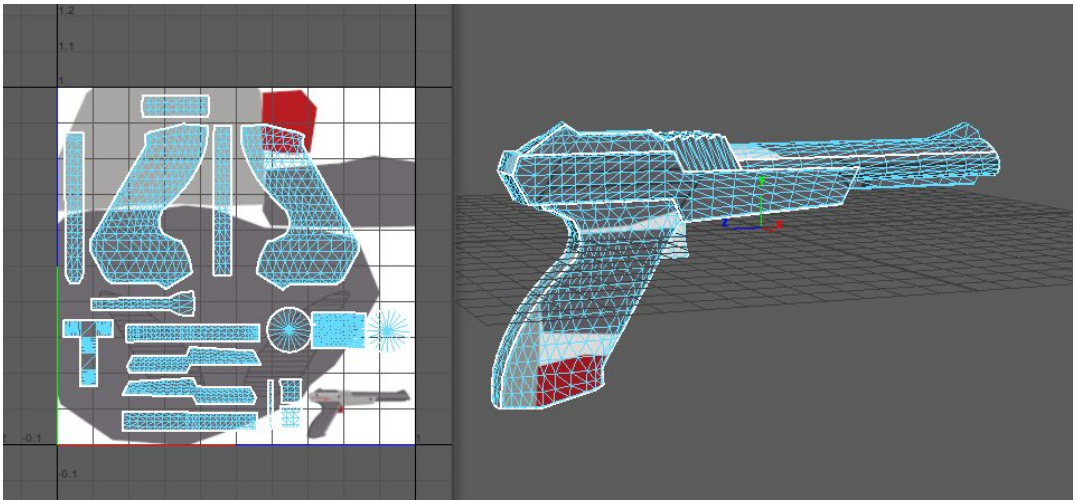
Every model was perfectly imported into the scene without any difference from the Maya version. Despite the ground having 80000 faces, there were no notable performance issues.

Below is the expected Maya result vs. the final result:

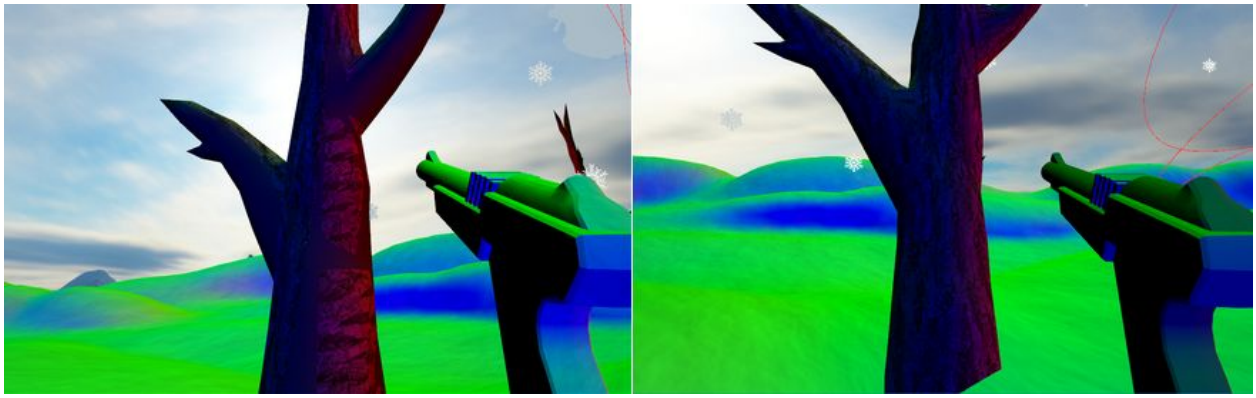
2.c) Texturing

Methodology

UV Unwrapping: The first step in the texturing process which is basically cutting the 3D model into surfaces and projecting 2D images on these surfaces according to the artist's vision.



UV Mapping: The 3D software generates an obj file which includes the UV coordinates. These UV coordinates should match on both 3D software and our project.



The screenshot on the left is an example of wrong uv mapping. The v-coordinates were flipped which can be fixed easily using (1-v) or by just flipping the texture image vertically to get this result (this problem appeared because we were using different types of models: primitives and objects).

VAOs and shaders: In our project we have two different types of models: primitives like spheres and cubes and objects like the duck and trees. These models have different VAOs for example primitives have color and does not have texture, conversely is the objects. We tried to use different shaders for the models and for the objects, but that was not efficient. So we made a shader that can contain them all (previous photos show loading the normals as colors).

Binding textures: After wrapping, mapping, and loading the textures there is only one task remaining: binding the textures before drawing.

```
duckTextureID = TextureLoader::LoadTexture("../Assets/Textures/duck_texture.png");
```

```
else if ((*it)->GetName() == "duck") {

    glActiveTexture(GL_TEXTURE0);
    GLuint textureLocation = glGetUniformLocation(Renderer::GetShaderProgramID(), "myTextureSampler");
    glBindTexture(GL_TEXTURE_2D, duckTextureID);
    glUniform1i(textureLocation, 0);
}
```

2.d) Skybox

Methodology

The skybox was implemented in order to give a more realistic feel to the game by giving the player the illusion that the environment is much larger than it actually is. The technology used in skybox is a cube map in which a special type of texture that is mapped on the cube. The image files are loaded by *stb_image.h*. The 3D texture coordinate used in order to sample from the cubemap is *GL_TEXTURE_CUBE_MAP*.

Initially, the texture object was created first, and then the cubemap enums (which represent the cube map faces) were loaded into the loop by using vector consist of the corresponding image file names. Afterward, the clamping mode assigned to each dimension of texture object was *GL_TEXTURE_WRAP_R*.

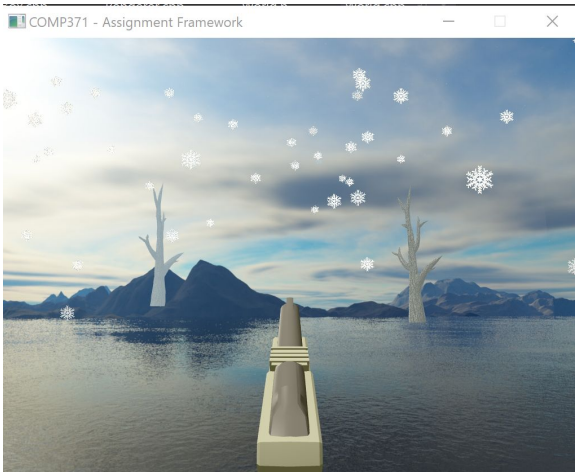
In the second step, the vertex and fragment shader was created. In the vertex shader, the incoming position was transformed by the world view projection matrix and the outgoing vector which goes to fragment shader, and the z-component overridden with the w-component to make sure that the final z-value would be 1.0. This way, the skybox would map to far-z and always fail the depth test. In the fragment shader, the *samplerCube* sampler was used to be able to access the cubemap.

In the final step, in the draw method of *World.cpp*, the skybox was drawn after all other models. The depth function is set to *GL_LEQUAL* to avoid the skybox being clipped. The vertex shader input was assigned and *glDrawArrays* called to draw triangles.

Optimization

By drawing the skybox at the end, the fragment shader would only run for the pixel encompassing the background of the scene and not the ones that are covered by the other models. Since the depth buffer is populated with all z-values by the time the skybox rendered, this gives us the ability to use early depth test and discard a fragment if it fails the test without executing the fragment shader.

The result is an enormous scene with a panoramic view representing the sky and mountains:



2.e) Shooting Bullets

Methodology

The shooting implementation was done in *Bullet.cpp*. Because a sphere is easier for calculating the collision, the *Bullet* is inherits from *SphereModel*. When the user clicks the mouse, it will trigger the function in *World.cpp*. The sphere will be drawn on the window, and the z-axis will be increased for every frame draw based on the function, *mPosition += mVelocity * dt*. The bullet will move further and further.

Alternatives & Result

Instead of inheriting from SphereModel, it is possible to create the sphere in the *Bullet* class. Also, it's possible the create the mouse trigger in *Bullet.cpp* rather than in *World.cpp*. However, it would create complexity in our program. To minimize the complexity, we desired to put the mouse trigger function in *World.cpp* and inherit the code from *SphereModel.cpp* to *Bullet.cpp* for the same result.

2.f) Bullet-Duck Collision

Methodology

Bullet-Duck collision was implemented using collision spheres. The duck was given a "radius" as if it were a sphere, and once every update tick the locations of both the duck and (each) bullet were compared against one-another. If the distance between their center points (calculated using the Pythagorean theorem) was less than or equal to the sum of their radii, a collision was considered to have occurred.

The duck's given radius was chosen to be large enough that more than 80% of the model would be inside the sphere, giving the players the ability to hit the duck even if they're off by a little bit in most cases. This was an intentional choice, as players of most games are more likely to complain about a game imbalance that is not in their favour rather than one in their favour (if the player felt they should have hit

the duck and didn't, they would likely complain, however if the player felt they should not have hit the duck and did, they are unlikely to complain).

Alternatives

There are a great many ways to do collision, and the most common alternative to the chosen way would have been bounding boxes (or in this case, rectangular prisms). If this method were used, the collision would have been far more accurate, as the duck's model is longer than it is wide, and especially longer than it is tall. However, the reason this method wasn't chosen was due to time constraints. It is far quicker and easier to implement spherical collision than it is to implement bounding boxes, as bounding boxes require a comparison for each corner. In a three-dimensional game, that amount of corners doubles as the z-coordinates must be checked in addition to x & y.

A different type of alternative would be to do incremental collision detection. Rather than checking every single Bullet in the list against the location of the duck, we could first check every single Bullet *near* the duck by doing something akin to the binary search through the Bullet list. The reason this method would work is because the bullets are entered into the list chronologically (the first bullet in the list would be the oldest bullet, and the last bullet in the list is the newest). Therefore, if bullet n/2 is already further away from the player than the duck, then there is no reason to check every single bullet from 0 to n/2. This could drastically save computation time if we had multiple players, or a machine gun style of gun that shot many bullets per second.

Results

In the end, the method that was chosen was chosen for two reasons: time and simplicity. The constraint of time was the reason behind the spherical type of collision. Also, due to the fact that we use a single-click fire gun (it only shoots a bullet once per click, so the amount of bullets created is limited by the speed the user can click the mouse), it was decided there was no need to add additional logic beyond the basics. The collision sphere of the duck can be shown in-game by pressing the [C] key. This is a SphereModel that follows the exact position of the duck, with the same radius given to the duck.

2.g) Duck Movement

Methodology

The duck's flying path implementation was done by reusing the assignment framework, with the duck replacing the sphere and using the splines to make the path smooth. The duck's flying uses key frames. We also keep a sphere at exactly the same location of duck, however the sphere and *Gl_LOOP* line is hidden unless the [C] key is pressed. When the duck is flying, we have to make sure the head of the duck is facing the next direction. To do that, we use this function to make sure the duck won't rotate itself:

```
if (firstKeyAngle > secondKeyAngle)
{
    if (firstKeyAngle - secondKeyAngle > (secondKeyAngle + 360) - firstKeyAngle)
    {
        secondKeyAngle += 360;
    }
}
else if (firstKeyAngle < secondKeyAngle)
{
    if (secondKeyAngle - firstKeyAngle >(firstKeyAngle + 360) - secondKeyAngle)
    {
        firstKeyAngle += 360;
    }
}
```

Alternatives & Result

To make sure the duck is facing the same direction and won't rotate itself, we could also use trigonometry or we could find the lootAt vector for duck model to do the same thing. However, it would add complexity and get the same result.

2.h) Duck Death

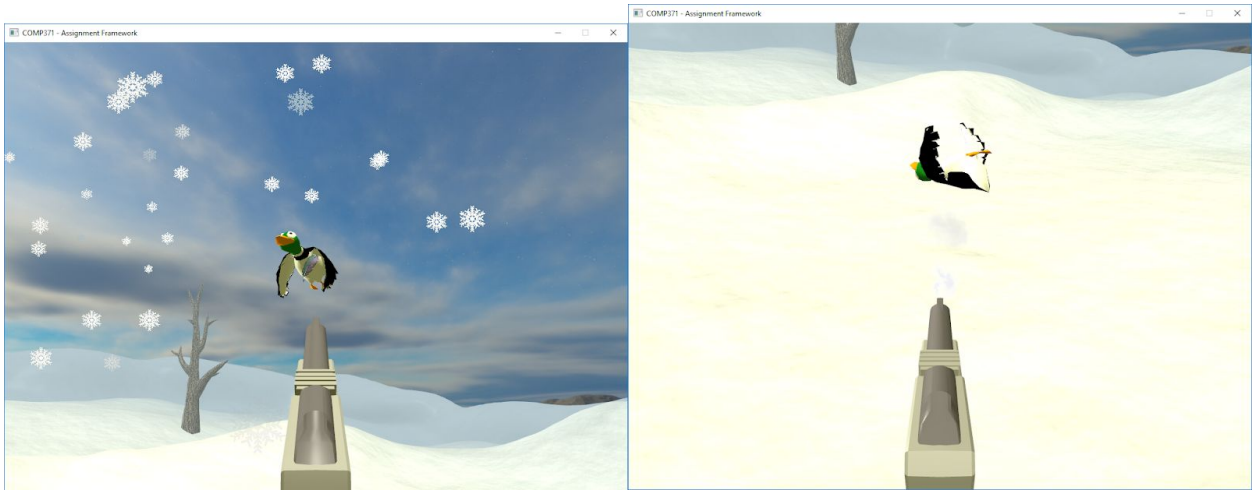
Methodology

Duck death's implementation was heavily based on time constraints, as it was the very last feature added to the game (though the animation for it had already been implemented far earlier). As a result, there are a few broken unspoken rules in the implementation of duck death (member variables being public for ease of access).

Duck death is implemented quite simply. When a collision is detected between a bullet and the duck, there is a boolean value assigned to the duck representing its death that is set to true. Additionally, the duck's position and rotation at the time of collision is recorded, so that the animation function's interpolation between two key points no longer affects it. A delay is set so that the duck freezes in place for a moment. Once the delay runs out, a rotation is applied to it so that it appears flipped upside-down, and the duck's position is affected by a vector with negative y multiplied by a timer that is slowly incremented (so that the duck falls straight down)

Alternatives & Result

Most of the alternatives involve different ways of coding the same final result. The member variables could be set to private and accessed with getters and setters, a new spline path could be used rather than a simple downwards vector, and so on. All of these methods felt superficial in light of how little time was left to implement the feature, and so we settled on the fastest method of implementation that gave the result we desired.



2.i) Particle Effects: Snow

Methodology

Snow particles were implemented to add more reality to the scene we have drawn. The snow particles are not emitted from a single position as the other particles. They are emitted from different points in the xz-plane.

```
newParticle->billboard.position = vec3(EventManager::GetRandomFloat
    (-20.0f, 20.0f), 20.0f, EventManager::GetRandomFloat(-20.0f, 10.0f));
```

Snow particles have different sizes, angles, and lifetimes, and they are set to always face the camera.

Alternatives
An alternative or enhancement could be using different textures for the snow (rather than a single one) that would make the particles more real. However for the cartoonish scene we have, snow flakes were just fine.

2.j) Particle Effects: Temporary Effects

Methodology
Both the duck feathers that spawn on bullet-duck collision as well as the smoke that appears to come out of the tip of the gun after shooting were implemented in the same way. Both are attached to a tiny cube that is set to the position vector 1000,1000,1000 in order to not appear on the screen. Both then have a boolean that is checked in order to see if the particles should be appearing, and if so, the position of the appropriate cube is moved to the appropriate position for a set duration, and then moved back away once the timer is finished counting down to zero.

In the duck feathers' case, once a collision occurs, the cube's position is set to the location of the duck at the time of the collision. Once that timer reaches zero, the cube's position is sent back to 1000,1000,1000. The timer is maintained in the main update function of the world, rather than the update function of the cube (because the two cubes have different types of timers).

In the case of the gun smoke, there are two timer variables. Once the player clicks, both the cube's location timer as well as a delay timer are set to a non-zero number. The delay timer decreases first, and once it is zero, the other timer may then decrease. The cube will remain at position 1000,1000,1000 unless both the delay timer is at zero and the location timer is greater than zero. If that is the case, the cube is set to the position of the camera, offset by the camera's lookAt vector in a way that makes the smoke emitted from the cube appear to come out of the tip of the gun.

Alternatives
An alternative would be to set the duck feathers particles to always be the location of the duck and the gun smoke particles to always be the location of the gun, and simply only begin emitting particles when the above conditions are met. However, given that we already had working cube emitters in the code from Assignment 2, it seemed reasonable to not attempt to reinvent a new method when the current method achieved the same result.

Results
While conceptually it makes sense that two cubes always emitting particles would have a reduction in performance compared to emitters that are only active on the trigger, the actual performance drop is

negligible. The reason for this is that the cubes are placed so far away when inactive that the frustrum culling prevents them from being drawn.

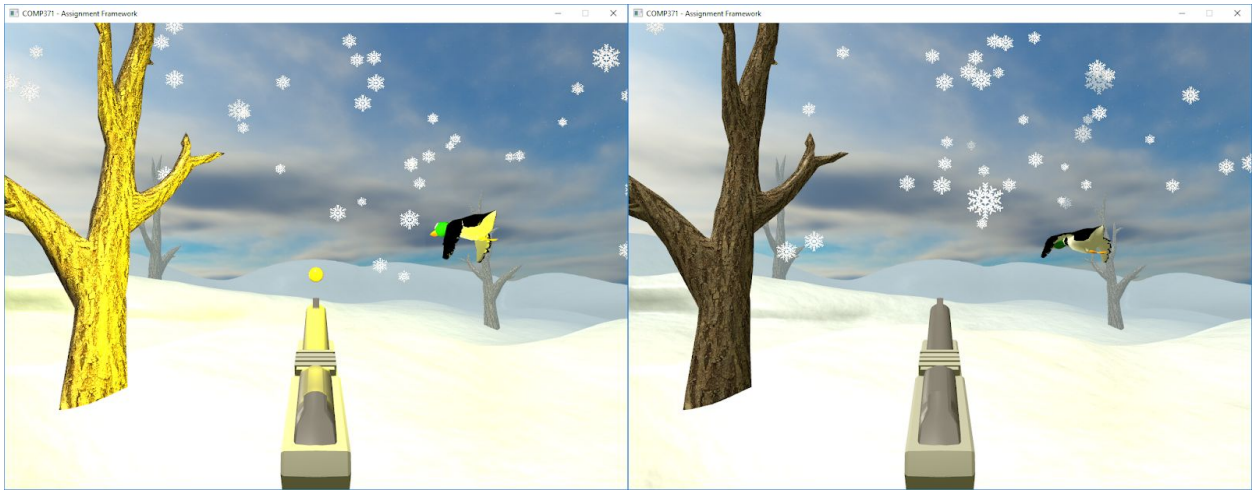
2.k) Lighting

Methodology & Results

The basics of lighting were implemented as [*color = color * (ambient + diffuse) + specular*]. This is nearly exactly the same as the general lighting calculations from Assignment 3, with the aesthetic choice of the specular lighting being based on the light source's color rather than the texture color of the object being lit. With the basics in mind, the lighting in Team U's Duck Hunt is comprised of three sources of light and one additional aesthetic tweak. Sunlight is split into two of those sources as well as the aesthetic tweak, while the last source of light is the flash from the gun when shooting.

Sunlight, as mentioned, is implemented through the combination of several methods. The first is a simple directional light shining down on the scene, angled so that the player sees a bit of the specular shine. The second is a backlight, which is another directional light that is opposite in x & z coordinates of the regular sunlight. However, this light's ambient factor is completely ignored. Finally, sunlight is finished through the use of a static minimum ambient value. All models' ambient material coefficients are compared against this value, and the higher value between the two of those is used as the ambient, so that the models are always fairly well-lit even on their undersides.

Finally, the gun light is implemented in a very similar manner to the temporary particle emitters. The light is placed far off in the distance where none of it will be rendered due to attenuation (because it's a point light). When the player shoots, the light's position updates to the camera's position modified by the camera's lookAt vector, so that it appears to light up at the tip of the gun. In order for this to be a little more realistic visually, the light is actually placed slightly above the gun, otherwise the light would not affect the part of the gun that the player can see.



2.l) Fog

Methodology

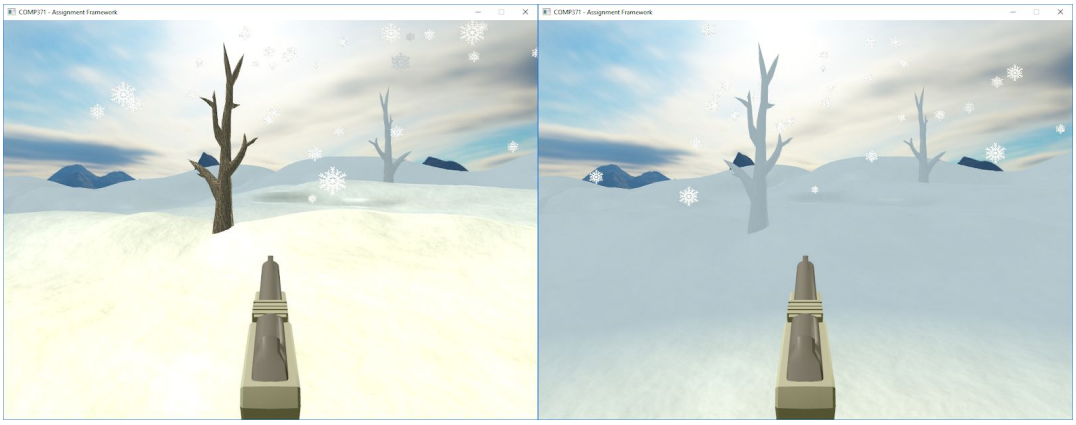
The distance fog implemented is a linear distance fog. In order to do this, both "start of fog" and "full fog" distance values were needed. The fragment shader takes both of these values and compares the location to both of these points, adjusting the color based on that. The specific function is:

$$f = (\text{distance}_{\text{from\_camera}} - \text{distance}_{\text{fog\_start}}) / (\text{distance}_{\text{full\_fog}} / \text{distance}_{\text{fog\_start}})$$
$$C = C_{\text{fog}} * f + (1 - f) * C_{\text{regular}}$$

The reason for a start of fog distance is because otherwise the fog begins immediately from the camera. Even if the full fog distance is set very far, everything in the scene is rendered slightly off color, and it doesn't look great.

Alternatives

Exponentially calculated fog is another approach to adding distance fog, however just like calculating linear without a start of fog distance, exponential fog begins at the camera. Even if the difference is minimal, everything is drawn slightly off color, and it doesn't look great. A possible alternative is to simply give exponential fog a start of fog distance. However, that wasn't considered until after the fog was already added, and at that point it would have been impractical to go back and change it when time could have been spent on other features.



3) Discussion

3.a) Overall Methodology & Results

In general, most of the decisions made in how to implement a feature revolved around one thing: efficiency. With the time constraints of this project, we needed to make sure a feature was easy enough to implement that we wouldn't be spending all of our time on it. We also needed to make sure it was functionally efficient, as a methodology that drastically lowered the FPS made it less desirable. The animation is a good example of this: originally, the animation was calculated more smoothly, however the CPU couldn't keep up and thus we changed to a simpler method. This is also the reason we decided to keep only a single duck on screen; more ducks would have significantly impacted the FPS (this worked out, as the original Duck Hunt only had one duck on the screen at a time anyway).

The results turned out great. Duck Hunt was given a very winter sort of feel, and every aesthetic choice (the ground, the trees without their leaves, the snowflakes, the distance fog fading to a wintry sky blue color and then skybox) made contributes towards this while still capturing the feel of the original game's artstyle thanks to the texturing. Overall, we feel we've been very successful with this project given the time constraints involved.

3.b) Future Work

There are several missing features that we would have liked to implement, had we the time. The most notable of these features are:

- The UI: our idea would have been to create several in-world scoreboards textured to display the score, the amount of remaining bullets, the remaining ducks, and the current level (the functionality of all of which would have also been added)
- Shadows: the shader for the shadows is in (and unused), but none of the rest has been implemented. The idea was to render the scene from an orthographic view of the sunlight and render each visible point's depth to a depth map, and then pass this depth map to the normal

shaders when rendering the actual screen. After multiplying a visible vertex by the light's view matrix, we can compare the depth to the depth map and not affect that vertex by sunlight if the depth is a higher number than the one in the depth map. While this is only the first step (it causes shadow acne), we would have needed to do more research for the next step.

- Gameplay: the idea was to make the game mechanics much more similar to that of the original Duck Hunt. (three bullets per duck with a duck respawning and bullets restocking if a duck was killed or got away, new levels or game over once all of the ducks in the level either have been killed or have escaped, etc).

3.c) What was Learned

3.c.i) Alan

In gun shooting part, I've learned how to implement the shooting function , how to calculate the matrix and adjust location. I learnt how to manage the frame to make it more effectively. Also how to texture the bullet color.

In the duct animation part, I've how to relocation the model and how to calculate the angle when the object is in animation.

Although, it is not in the program, I also help my teammate for code management in Github, such as merging and conflict, and debugging. I believe what I learnt in this class will be beneficial for my future career.

3.c.ii) Amrou

The most valuable skill that I have learned through the project, is texturing. Texturing has two sides: the artistic side and the programming side. I learned a lot about UV unwrapping and UV mapping. I understood the VAOs and the shaders in a much deeper while implementing my task. I learned how math, physics, and art together can work together to create a beautiful scene.

I learned more about camera coordinates and camera movement in general since I had to clamp the camera to the ground and limit the angles of vision. Implementing also a particle system has also taught me a lot about particles and billboards.

In general the project gave me a deeper understanding of many different features of opengl and computer graphics in general. I feel now that I understand the bigger image, compared to the assignments in which we were doing minimal adjustments without understanding the whole image.

3.c.iii) Ghislain

Making and integrating the Models was a straightforward work, but that allowed me to get a better grasp at the process of creating models in OpenGL. The Animation part is where I learned the most.

Although it was too late to do the changes, I've learned how important it is to let the GPU do the heavy calculations. Reaching 4 FPS with a model of 2 thousands faces and 4 moving bones was far slower than I anticipated, and I don't see 30 FPS being possible with a human model of over 10 thousand faces and over 30 bones rotating in all directions and making use of IK Handles, using only the CPU. This low FPS forced me to also get creative with performance optimization, which was certainly an interesting experience that could help in other programming fields.

3.c.iv) Maryam

I've learned how to load images in framework and map it to 3D texture coordinate. Besides, I gained the better understanding of how the GLSL shader languages work and how to debug the shaders.

I studied about how GPU's has optimization mechanism to boost the performance and I used early depth test technique in my task to render skybox in the most efficient way.

3.c.v) Robert

Despite it not being implemented, the thing I learned most about whilst doing this project was the concept behind implementing shadows, and one of the common effects when shadows are first implemented: shadow acne. Aside from that, I solidified my knowledge a great deal about implementing lighting into a scene, as well as learned how to implement multiple lights at once (something I was previously very uncertain about).

I was already familiar with a lot of the techniques I used to implement other features, such as collision or moving something far off-screen so as to not display it on-screen (eg. with the particle effects and gun light). While I was already very familiar with C++, it had been a long time since I used openGL at all, and therefore relearned a great deal about that (especially how to send information to, between, and from the shaders).

Most importantly to me: I was forced to learn a great deal about github in order for our teamwork to be smooth and for there to be less conflicts in coding. The reason I state that this is the most important to me is that I generally don't code with either C++ or openGL; the game I'm publishing is made using C# instead, without shaders. However, I suspect I'll need all of this information for a future game, so it's all extremely valuable in the end.

4) User Manual

4.a) How to Compile & Run

1. Download the project from this link: https://drive.google.com/open?id=1MjV64aspr4_u61Q9jQaINuRYkBnFet8_
    a. Alternatively, visit the GitHub link: https://github.com/alanfok/Comp371DuckHunt
2. Navigate to the *Duck Hunt/Bin* folder and double-click the *Assignment1-Release.exe* file
    a. Alternatively, navigate to the *Duck Hunt/VS2017* folder and run the *.sln* file to see the source code and solution. You may have to retarget the solution for it to run.

4.b) User Controls

Basic Controls
Aiming and shooting is done with the mouse (left click to shoot)
Movement keys (must be enabled by pressing [M]) are [W][A][S][D]

Functional Controls
[M] - toggles movement (set to false by default)
[X] - toggles the ability to kill the duck (set to false by default)
[C] - toggles visibility for non-scenic objects, eg: collision sphere, particle cubes, pathing spline (set to false by default)
[B] - toggles ground visibility (set to true by default). Used to show the skybox
[N] - sets the duck's *dead* attribute to false (respawns the duck)
[F] - toggles fog mode (the default setting is less fog)

[7] - sets the duck's standard animation to flapping (the default setting)
[8] - sets the duck's standard animation to dead

Camera Controls
[1] - sets the camera to standard mode (the default setting)
[2] - sets the camera to overhead mode