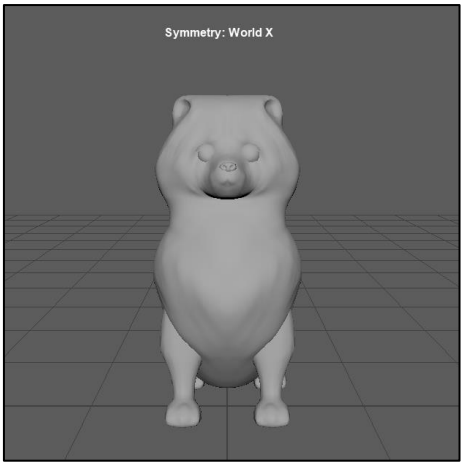
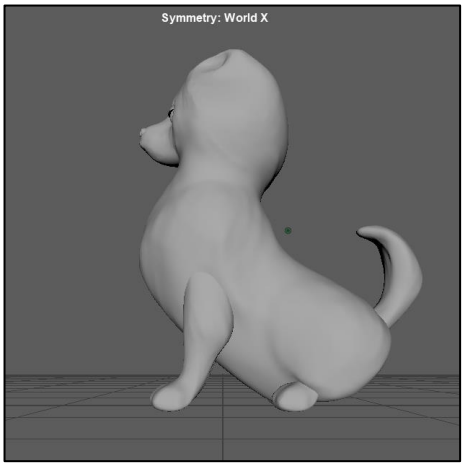
CSU44052 – COMPUTER GRAPHICS

ASSIGNMENT - MIDTERM REPORT

CONTENTS

Megan Whelan Dalton | 16324322 | whelandm@tcd.ie





THE MODEL

DESCRIPTION

For my model, I created a small Pomeranian style dog in Maya in a sitting position. I felt this pose was stationary enough to easily implement but also all enough options to create a hierarchy and animations.

PROCESS

During the modelling process, I started with a cube and extruded the faces to create a basic blocked body shape. I then manipulated edges and vertices while alternating between the cubic view and the smoothed poly view whiling forming a more natural shape. While working on areas that were on both sides of the model (such as the hips and ears) I turned symmetry on, on the objects x-axis, to speed up the process as I only needed to shape one side. I then modelled paws, and tail as separate parts, followed same method. For the nose, eyes, eyelids, used spheres instead of cubes, less manipulation needed.

Once I had basic shape made, I increase subdivides (Smooth), which increases the number of polygons in the model, so it resembles the poly view. I used quite a lot of subdivides to allow me to create very small details during sculpting. Once I had enough polygons to create natural strokes, I lifted and pushed back the polygons to create facial features and stylised illusion of fur (though this will also be later improved in textures). I liberally used the smooth tool to disperse any harsh lines to more natural appearance rather than clinical smooth surfaces before sculpting. Once finished with sculpting I decreased the subdivides (Reduce), this ensured less vertices to be loaded and stored by the program. I repeated this process again for the limbs and tail.

HIERARCHY

Before exporting the model to the '.dae' format, I separated it into a one to many hierarchy, and set each individual components' pivot to the origin to make transformations with OpenGL easier. The hierarchy for the model is as follows:

- Body (Root)
    - Tail
    - Paws (Front and Back)
    - Eyes (Left and Right)
    - Eyelids (Left and Right)

For the purpose of the midterm report I only exported one object from the hierarchy separately, the tail, as it was the only child I was planning on animating. However, for future features I plan to have each of these separate. The child mesh was loaded into a separate VBO to the parent body, making use of ASSIMP to load the mesh data into a MeshData object comprised of a series of vertices. As both of these objects had their pivots set to the origin there was no need for further manipulation or translation in program.

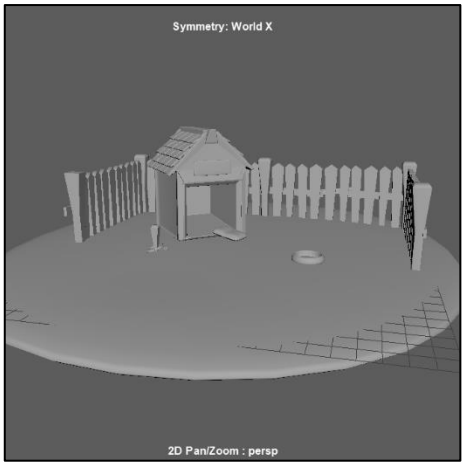
THE SCENE

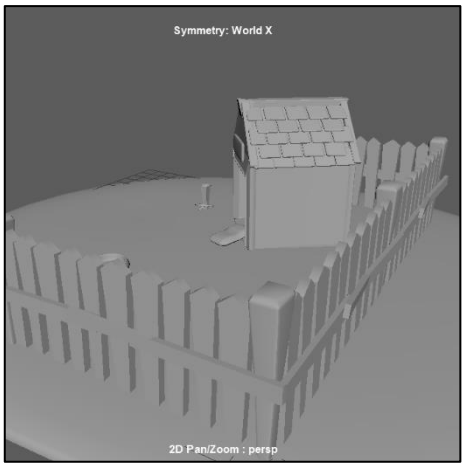
DESCRIPTION

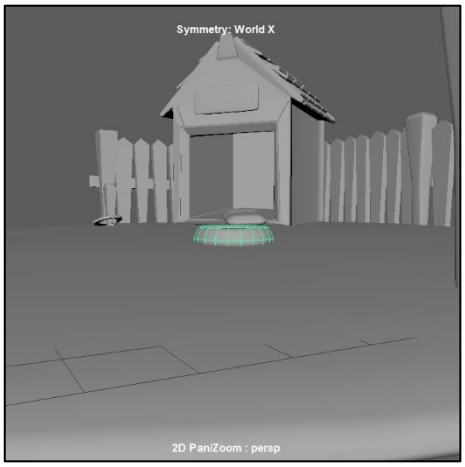
The scene I chose to construct was a simple, stylised back garden, with a kennel, picket fence and various dog-related items. I modelled each of the items individually following a similar, if not the same, process as that of the dog model but skipping the sculpting process. I could have added wood graining on the on the fence and kennel walls, however I will probably implement this effect with the texturing process in the future. If I feel the texture alone aren't enough detail, I can go back and add this in. I will also add in small clusters of grass scattered about to create variation and break up the monotony of texturing.

OBJECTS

- Picket Fence
- Kennel
    - Roof Tiles
    - Sign
    - Pillow
- Stake
    - Chain Links
- Spiked Collar
- Dog Bowl







THE CAMERA

I chose to implement an FPS (first-person shooter) style camera, that allows the user to "walk" around the scene by using the mouse and WASD keys. This camera has a fixed value on the y-axis that never changes giving the illusion of the height for the user or player.

MOUSE TRACKING

I use the lookAt() function to create the view of the world that serves as a the camera. This function takes both the position of the camera, the target as defined by the mouse movement as well as an up vector. The target or the focus of the view is determined by the position of mouse, within the window through the glutPassiveMotionFunc(). In this callback the mouse function takes in bot the x and y positions of the pointer within the window. The previous x and y values are saved for calculations of the direction, for this direction the previous mouse values are subtracted from the current values and based on it's relation to 0 the target position on the respective axis is moved by a small increment.

```
    if ((mouse_x - old_x) > 0)  // moved right
                target_x += 0.1f;
else if ((mouse_x - old_x) < 0)    // moved left
                target_x -= 0.1f;
     if ((mouse_y - old_y) > 0)  // moved up
                target_y += 0.1f;
else if ((mouse_y - old_y) < 0)    // moved down
                target_y -= 0.1f;
```

I have also implemented a range check on the mouse positions so that if it gets out of the defined boundary (within 20 pixels of the edges of the window border) it will be warped back to the center of the window. This check called the glutWarpPointer function to do so.

```
          // range check
         if (mouse_y < 20) {
        mouse_y = height / 2;
 glutWarpPointer(mouse_x, height/2);
                }
     if (mouse_y > height - 20) {
        mouse_y = height / 2;
glutWarpPointer(mouse_x, height / 2);
                }
          if (mouse_x < 20) {
         mouse_x = width / 2;
 glutWarpPointer(width / 2, mouse_y);
                }
      if (mouse_x > width - 20) {
         mouse_x = width / 2;
 glutWarpPointer(width / 2, mouse_y);
                }
```

WASD INPUTS

The camera motion is controlled by the WASD keys; W to move forwards, A to move left, S to move backwards, and D to move right. To create this motion in the relation to the keys, I make use of a forward vector (vec3), the up vector (vec3), and a defined speed (float) of the motion. The forward vector is calculated by taking the camera target (which is calculated by the mouse position, as explained above) and subtracting the camera position to get the distance from where the player is to where I would like it to be.

```
forward_x = target_x - camera_x;
forward_y = target_y - camera_y;
forward_z = target_z - camera_z;
```

If a W keypress is logged the forward vector is simply multiplied by the speed and this value can be subtracted from the current camera position to “walk” towards the target the target, conversely for an S keypress the value is added.

```
if (key == 'w') {    // Move camera forwards
        camera_x += forward_x * speed;
        camera_z += forward_z * speed;
        target_x += forward_x * speed;
        target_z += forward_z * speed;
```

By also adding these values to the camera target we can allow for continuous motion in all directions. For sideways motion in relation to the target the cross product of a combination of the Up Vector and Forward Vector must be used. In the case of an A keypress, indicating the desire to move left, I used a Left vector that is the cross product of Up*Forward, whereas the right motion the cross product used is Forward*Up. The vector is the multiplied by the speed and as before added to the camera position and camera target.

```
            if (key == 'a') {    // Move camera left
vec3 left = cross(vec3(forward_x, forward_y, forward_z), up);
                camera_x -= left.v[0] * speed;
                camera_z -= left.v[2] * speed;
                target_x -= left.v[0] * speed;
                target_z -= left.v[2] * speed;
                            }
```

OTHER CHOICES

I disabled the mouse pointer from appearing so motion would feel more natural and the user would not be distracted by it.

CURRENT FEATURES

KEYBOARD INPUT

Actions within the window are all controlled through keyboard input by the user. These inputs are tracked by a series of call-backs defined within the following functions:

- glutSpecialFunc(arrows): for special keys such as arrows
- glutKeyboardFunc(keypress): general keyboard input (ASCII)
- glutPassiveMotionFunc(mouse): mouse tracking

These exact controls will be discussed in more detail below, defining what keys do what and my reasoning for their implementation or use in the future.

TRANSLATIONS

Camera translations are controlled by the WASD keys, as discussed in detail in the Camera section above. These change the view within the world, updating the camera position vector and the camera target vector.

Object translations are implemented with the arrow keys which require the glutSpecialFunc() callback rather than the glutKeyboardFunc. This will probably be removed in my final project as it isn't a very natural looking movement. Possibly use it for another object ie. moving food bowl towards dog to trigger an animation within a certain range of proximity.

```
if (key == GLUT_KEY_DOWN) {
   translation_y -= 0.1;
                }
 if (key == GLUT_KEY_UP) {
   translation_y += 0.1;
                }
if (key == GLUT_KEY_LEFT) {
   translation_x -= 0.1;
                }
if (key == GLUT_KEY_RIGHT) {
   translation_x += 0.1;
                }
```

ROTATIONS

The model can be rotated on the x-axis using the ',' and '.' keys, the comma rotating it clockwise and the period rotating anti-clockwise. This is simply to illustrate the object hierarchy working, showing that both the dog rotates while the tail rotates and wags, as described below.

ANIMATION

The tail of the dog is animated to "wag" on the keypress of the 'x' key and stop on the 'z' key.

```
if (key == 'x') tail_rotation = 30.0f;
```

The tail rotates 15 degrees of the axis on either side of the axis, once it hits this point it switches directions and begin to rotate 15 degrees in on the opposite side of the axis, giving the illusion of the tail wagging.

```
tail_z += tail_rotation * delta;
if (fmodf(tail_z, 15.0f) > 0) tail_rotation = -30.0f;
    if (tail_z/15.0f <= -1) tail_rotation = 30.0f;
```

FUTURE FEATURES

RIGGING

- Rig dog, with joints in head, ears, along paws, and in the hips and base of tail. This is a simple enough skeleton that should allow for more natural looking animations to be implemented.
    - Such animations could be the hips wiggling while the tail wags, or paws tapping.
- The rigging of the head and ears is to allow for the player tracking discussed below.

PLAYER TRACKING

- With rigging in both ears and head, this could allow for rotation in relation to where the camera is currently positioned, within a reasonable degree (avoid exorcist).
    - Based on location of camera animate the head or ear, set boundaries for certain reactions, so further away only ears would track, but once within boundary the head can turn too.
- A simpler version of this could also be have the eyes rotate to mimic the gaze following the player rather than the full head.

TEXTURING

- As not all objects in the hierarchy look very pleasing with the flat shader colouring, certain objects will have their own texture.
- Certain hierarchical objects will have multiple textures within, one put each member of the hierarchy.
    - For example. The kennel would have separate texture for the roof tiling than the walls.
- For smaller or less visible objects, keep flat shader colouring as less noticeable,

LIGHT

- As an outdoor scene, without any artificial lights will have one light source.
- However, I would need different objects within a hierarchy to make use of different shaders and styles of lighting.
    - For example, the eyes and nose of the dog would need to use a glossy shader or anisotropic to give an almost wet appearance, while the rest of the body would need to appear more matte with a diffuse approach.
- Apply specular maps to the kennel, to give the appearance of the roof tiles and wooden walls reacting differently to the light as result of their material composition.

EXPAND HIERARCHIES

- Further expand on the object hierarchies to allow for implementation of the aforementioned features.

OBJECT COLLISION

- Implement object collision for all objects within the worlds, this would prevent the player from ghosting through objects. This can be achieved by setting up checks before camera translations are performed and stopping them if they are within an object boundary.
- Also implement around the fence of the scene in order to stop the camera from travelling out of bounds.

LINKS

VIDEO DEMO

https://www.youtube.com/watch?v=HWmFZuqFYmM&feature=youtu.be

MAIN.CPP

https://drive.google.com/file/d/1Pb9Px8xm1gfeNgwJWBv-abh1ajoX6tE6/view?usp=sharing