Zak Olyarnik
CS-432 Final Project

I. Introduction

This project is directly (and obviously) inspired by the Sonic the Hedgehog game franchise. It's set up as an "interactive animation": Sonic is constantly moving forward along a flat track, and the user can control his motion by using the arrow keys to switch "lanes" and by clicking him to jump. The motivation for this is to make Sonic collect the rings scattered along the track by passing through them, after which they will disappear.

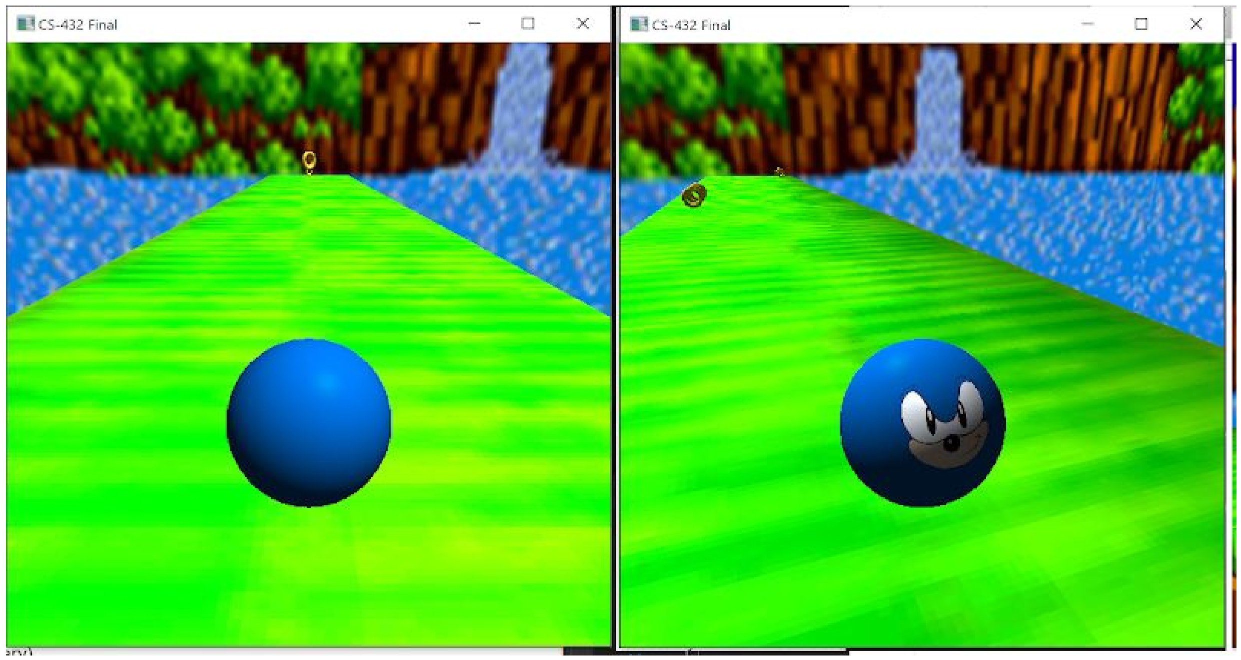
At the end of the track, there is a shuttle loop, and as Sonic approaches it, the camera will auto-switch to an optimal view to watch him roll around it. After the loop, the track will reset, sending Sonic back to the beginning, turning any collected rings back on, and even randomizing their positions.

The main camera is always centered on Sonic, but can be rotated to make it orbit around him to get a different angle. A third camera is also provided, and switching to it lets the user view the track in 2D, exactly like the classic games. Jumping and lane-switch are also implemented in this view, although since the camera is set up to be orthographic, it's hard to tell which lane Sonic or the rings are in when in this mode.

Nothing from the "advanced functionality" list is included; I simply ran out of time.

II. Users' Guide

A. Main View



In this view, the camera starts behind Sonic and at a given elevation, as a typical game camera would. Pressing the **z** and **x** keys will rotate the camera around Sonic, keeping it centered on him, which is also extremely typical game behavior.

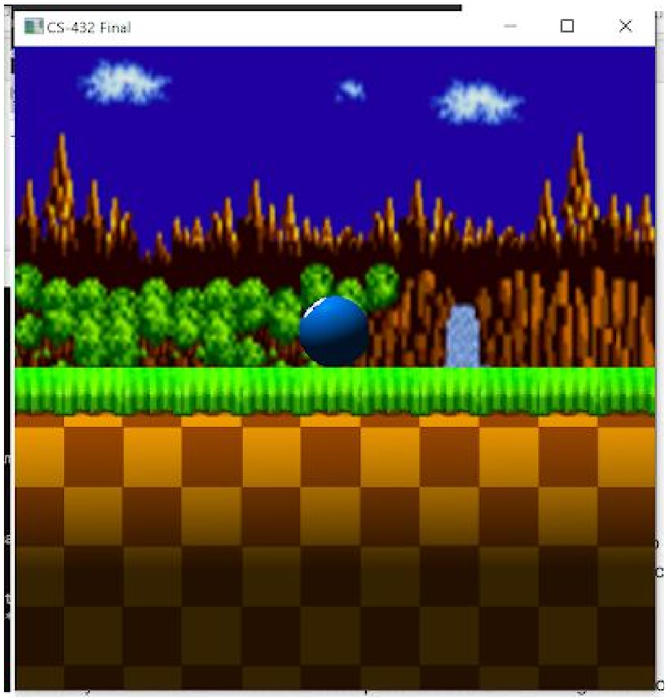
The track is divided into three imaginary "lanes", and using the **left** and **right arrow keys** will move him between the lanes. Checks are included in the code to always keep Sonic on the track. When an arrow key is pressed, a check of the relative z-positions between Sonic and the camera ensure that he will still move in the expected direction; i.e. if the camera is in front of him, as in the right image, pressing the right arrow key will still move Sonic to the right *on the screen*.

Clicking on Sonic will make him jump. This is accomplished using picking, so clicking anywhere else will have no effect.

Pressing the **spacebar** will switch from this view to the 2D View.

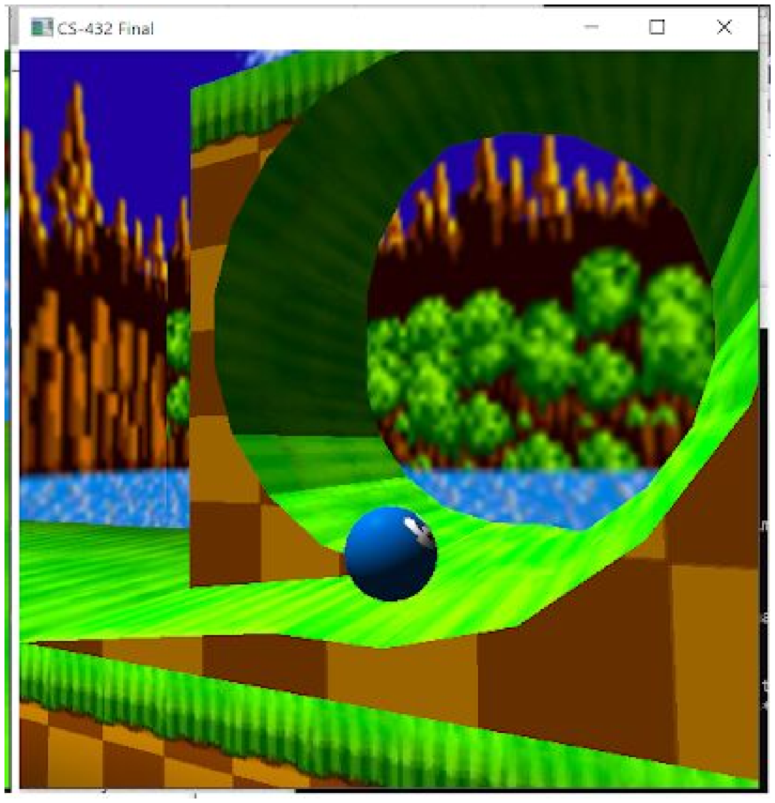
Collecting rings by passing through them will make them disappear. Move and jump to line Sonic up, and try to collect them all!

B. 2D View



In this view, the 2D camera is fixed and will continuously move to the side as Sonic does. The camera cannot be controlled otherwise. Switching lanes and jumping is implemented the same as in the Main View, but because the camera is set to be orthographic, it can be hard to tell what lanes the objects are in, and that Sonic is switching lanes at all. Pressing the spacebar will switch from this view back to the Main View.

C. Loop View



In this view, the camera is completely stationary, and is set to provide the best angle to watch Sonic rolling around the shuttle loop. User input is not read at all in this view. The camera will automatically switch to this view right before Sonic enters the loop, and will automatically switch back to the Main View (and reset to the default orbit position) after he exits it.

III. Technical Details

A. The Use of Timers

Since this is an “interactive animation”, all the key events are set to occur at defined times. The main function includes a timer which is always counting up, and several variables defining the exact frames when Sonic reaches the ramp onto the loop, the center of the loop, the end of the loop, and the end of the course. These were all figured out by trial-and-error with a stopwatch, determining “what looked good” once everything had been modeled and placed in the scene.

I figured out whether I had reached the correct timestep by using a cool modulus trick: by taking the current time (not resetting per animation loop) mod the total animation time, I’d get how far into the animation I was, and then compare that against the pre-defined event times.

B. Moving the Sphere

The sphere, Sonic, is constantly in motion, including both a roll around its x-axis and a z-transformation to move him down the track. Appropriate angle and distance values were determined to make the movement look realistic in the animation timestep, so it appears that he is moving at a good speed and without teleporting.

In order to control lane switching and jumping, more variables were necessary. After more trial and error, “max time spent moving” values were defined; i.e. “it takes this many timesteps for Sonic to move from one lane to the one beside it” or to reach the top of his jump arc. When a movement key is pressed, those numbers are set, and at every timestep afterwards, they count back down to zero. While they are not zero, translation functions on the sphere are called. When they reach zero, Sonic has completed his movement and will stop moving.

At certain key events (reaching the ramp to the loop, reaching the loop itself, exiting the loop), Sonic’s movement had to change. By using the loop’s physical measurements (see below), it was mathematically determined the appropriate y-translation to move him up the ramp, and the radial velocity and sin/cos numbers to make him roll around the loop.

A final point about the sphere is that I kept getting weird texture distortions at the seams, no matter what the position or orientation of my starting texture was. I had a crazy idea to try texturing it using a cube map instead, and since I really only wanted detail on the front face, I could use a solid-colored texture for the rest and have it look fine. This worked wonderfully.

C. The Cameras

I chose to modify the camera interface defined in all the previous assignments to convert it more into a typical game camera. This meant removing all of the free-motion aspects and instead making sure it was always centered on a target point. ...Almost. I started out with that target point being the center of the sphere, so that in the sphere’s draw call, I recalculated the position of its center point in world space, and then set that as the camera’s target, making it update every draw. The problem with this was that the camera would then “jump” when the sphere did, and that wasn’t the effect I was looking for. Instead, I ended up resetting the camera’s y-position and target to be fixed, and that made a jump move the sphere vertically on the screen. The orbiting motion of the main, player camera is controlled by a persistent angle that is updated whenever the rotation key is pressed, and then the camera’s position is updated using the sin and cos of that angle. I briefly looked at trying to rotate around the sphere in a different direction as well, but my tests kept resulting in gimbal lock so I abandoned that effort.

The 2D camera is set to ignore all movement input, and instead automatically update its z-position to keep it following Sonic in the same way the player camera does. The problems I encountered with making this camera orthographic had to do with the skybox (see below) and the picking formula. To make orthographic picking work correctly, the tested ray’s origin is the clicked point and its direction is the camera’s forward, which in hindsight makes perfect sense.

The loop camera is fixed and will ignore all player input. It automatically becomes active right before Sonic enters the loop, and will automatically switch back to the player camera (and reset to the default orbit position) after he exits it.

D. The Skybox and Skyplane

The skybox was created as in the last assignment, but using texture sprites ripped from the original Sonic game and then modified in Photoshop to make them tessellate a little better. After playing with camera settings and then doing research when it still wasn't working, I discovered that the concepts behind skyboxes don't work with an orthographic camera. My solution to make the 2D view still work was to just display a single image plane behind the rest of my geometry. Since my cameras all have an "is2D" flag, when I go to render the skybox and skyplane, I simply check this flag and don't render one or the other depending on the current camera's mode.

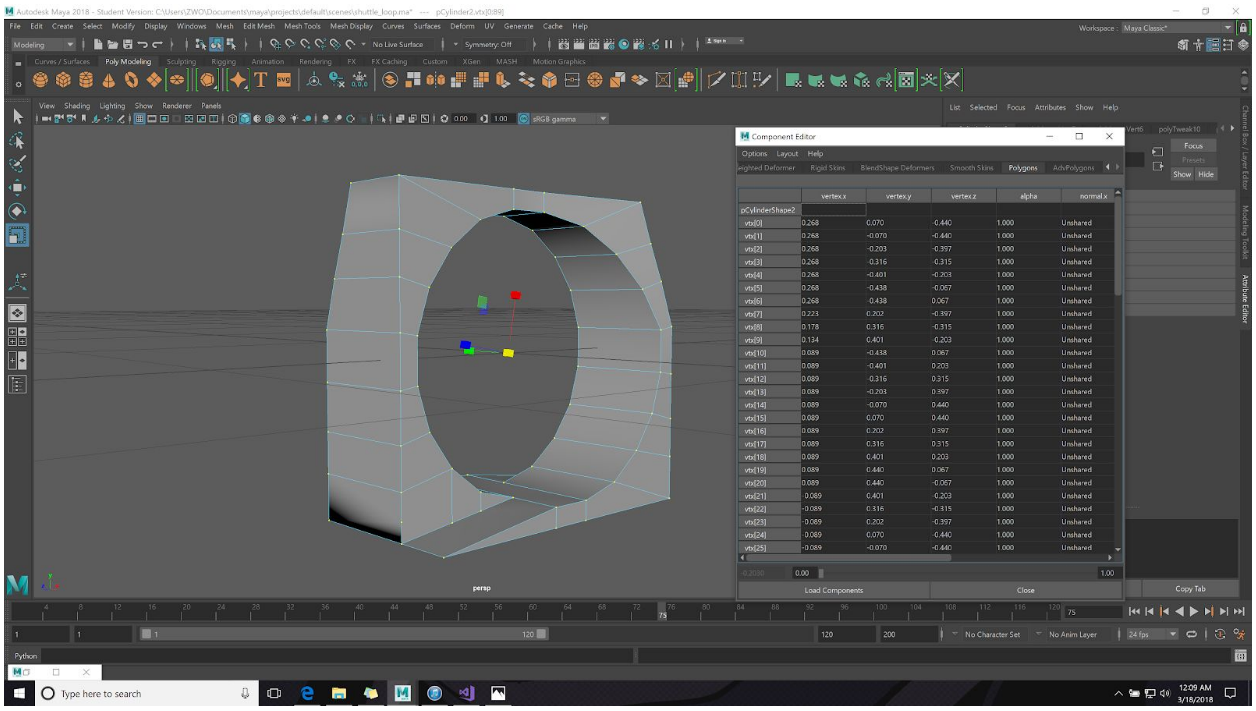
E. The Rings

I was able to define the rings parametrically, but in order to draw them correctly with glDrawArrays(GL_TRIANGLES…), it was necessary to also define "facets", sets of four points making up a face at a time. This ensured that the vertices were specified in the correct order when it came time to draw.

Ring collision detection is implemented using bounding spheres. I already had a way to get Sonic's center each draw call for the camera updates, and knowing each ring's radius allowed the creation of another detection sphere around each one. When it came time to draw each ring, I first check if it has already been collected (using a flag), and if so, exit without drawing or even doing the collision check. Otherwise, I do the check, exiting if I find a collision and setting the flag for next time. If no collision is detected, I draw the ring. A reset function (called at the end of the animation) turns all rings back on for the next run.

Rings are pseudo-instanced. Because I needed to be able to turn them on and off at will, I could not use true instancing as discussed in class. Instead, I use one set of positions and normals on the VBO, with an array full of vec4 positions, one for each unique ring. I simply adjust the model matrix in between draw calls in a loop. The ring positions are originally defined in groups of three based on the size of the track, and randomly placed in one of the three lanes and either on the ground or in the air. These positions are redefined in the reset, so that they appear in different places for each run of the animation.

F. The Loop



	vertex.x	vertex.y	vertex.z	alpha	normal.x
pCylinderShape2					
vtx[0]	0.268	0.070	-0.440	1.000	Unshared
vtx[1]	0.268	-0.070	-0.440	1.000	Unshared
vtx[2]	0.268	-0.203	-0.397	1.000	Unshared
vtx[3]	0.268	-0.316	-0.315	1.000	Unshared
vtx[4]	0.268	-0.401	-0.203	1.000	Unshared
vtx[5]	0.268	-0.438	-0.067	1.000	Unshared
vtx[6]	0.268	-0.438	0.067	1.000	Unshared
vtx[7]	0.223	0.202	-0.397	1.000	Unshared
vtx[8]	0.178	0.316	-0.315	1.000	Unshared
vtx[9]	0.134	0.401	-0.203	1.000	Unshared
vtx[10]	0.089	-0.438	0.067	1.000	Unshared
vtx[11]	0.089	-0.401	0.203	1.000	Unshared
vtx[12]	0.089	-0.316	0.315	1.000	Unshared
vtx[13]	0.089	-0.203	0.397	1.000	Unshared
vtx[14]	0.089	-0.070	0.440	1.000	Unshared
vtx[15]	0.089	0.070	0.440	1.000	Unshared
vtx[16]	0.089	0.202	0.397	1.000	Unshared
vtx[17]	0.089	0.316	0.315	1.000	Unshared
vtx[18]	0.089	0.401	0.203	1.000	Unshared
vtx[19]	0.089	0.440	0.067	1.000	Unshared
vtx[20]	0.089	0.440	-0.067	1.000	Unshared
vtx[21]	-0.089	0.401	-0.203	1.000	Unshared
vtx[22]	-0.089	0.316	-0.315	1.000	Unshared
vtx[23]	-0.089	0.202	-0.397	1.000	Unshared
vtx[24]	-0.089	0.070	-0.440	1.000	Unshared
vtx[25]	-0.089	-0.070	-0.440	1.000	Unshared

Rather than try to mathematically define the shuttle loop, I decided to use my background experience in 3D modeling to physically model the object in Maya. This was a simple task, and after finding a way to display all the vertex positions, I was able to manually define my “facets” for this shape as well. I also used the 3D model to make determining texture coordinates easier, although this also involved taking into account the applied scaling in the world and how to effectively repeat the texture to avoid stretching.

IV. Work Allocation

I worked alone on this project.

V. Citations

In order to use picking, it was necessary to retrieve the current viewport size (which is not the same as the window size). The strategy for doing so came from https://www.opengl.org/discussion_boards/showthread.php/166800-how-to-get-screen-size-in-openGL. In addition, in order to make picking work in the orthographic view, the definition of the ray we cast must change. The strategy for this came from https://www.reddit.com/r/gamedev/comments/1lpgju/is_raycasting_possible_with_an_orthographic/.

Zak Olyarnik
CS-432 Final Project

I looked up a formula to define the vertices of the rings. Unfortunately, a strict parametric approach caused problems when trying to render with glDrawArrays(GL_TRIANGLES…) because the vertices defined in this way don't follow the correct triangle winding order. Luckily, I stumbled upon an approach which defined the ring's surface in terms of "facets", which is exactly what I needed instead: http://paulbourke.net/geometry/torus/.

In order to make collision detection happen with the rings, I decided I needed a better approach than picking and casting dozens of rays between each ring and the sphere each frame. The easiest solution was to use bounding spheres, since the sphere was (obviously) a sphere already, and the spinning rings were always contained within individual spheres as well. The approach to doing this ended up being incredibly simple, but the reference I used to read up on the strategy was https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collison_detection.

The Sonic face texture came from https://www.pinterest.com/pin/394065036123430620. The Green Hill Zone textures were ripped directly from the original Sonic the Hedgehog 2D Sega Genesis game, and were found at https://www.textures-resource.com/dreamcast/sonicadventure2/texture/1004/. Both were modified in Photoshop to fit this project's needs, including building the different skybox planes from the sprite sheet.