COS 426 - Final Project Writeup

Collideoscope - by Noah Moss (ngb) and Evan Wildenhain (mew2)

https://github.com/ewilden/collideoscope/
https://collideoscope.github.io

Introduction

Goal

The “endless runner” is a fairly popular genre for small games due to its simple controls and intuitive mechanics. We chose this genre in order to have a project that could scale according to how on-schedule we were: assuming we could get the basic gameplay working quickly, we would then be able to iterate on the graphic style and other aspects of the design.

To arrive at a unique twist on the endless runner, we were inspired by the idea of setting the runner inside a kaleidoscope. Kaleidoscopes achieve very cool-looking visuals[1] via reflections, rotations, and other simple 2D transformations, so we thought this would be a good way to harness the ideas we learned in earlier parts of the course. In addition, making the player feel like they were inside a kaleidoscope would require a nontrivial amount of work on the geometries, textures, and materials of the scene.

Keeping this simple, but compelling, vision in mind—endless runner inside a kaleidoscope—we successfully implemented a basic endless runner inside a tube, then iterated on the visuals and gameplay (including mirroring effects, the barriers the player would have to avoid, the difficulty changing over time) until we arrived at a relatively-polished product that we were happy with.

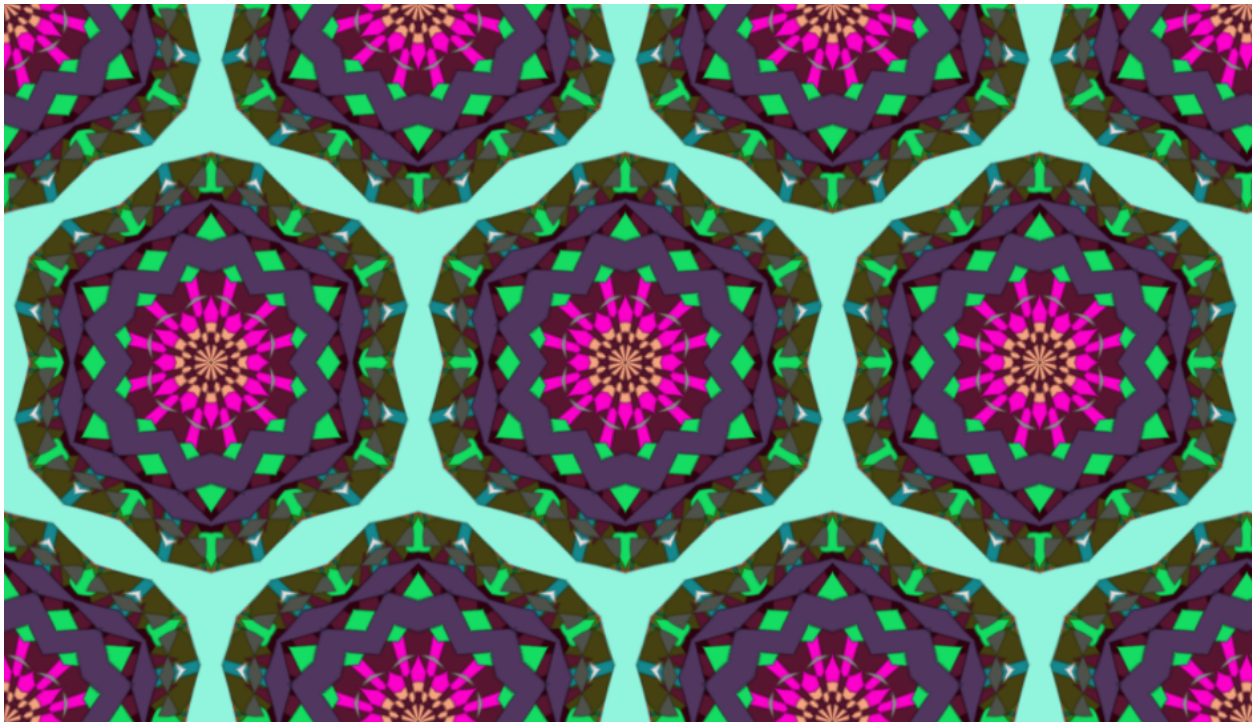
Previous Work

There is a lot of prior art for endless-runner style games. The earliest one we remember playing is *Canabalt*[2], but from the beginning we intended for *Collideoscope* to be a 3D-style runner into the page. As we started sketching out what we imagined the basic gameplay to look like—the player traveling down a tube, avoiding cut-out sections of kaleidoscopic barriers as they went—we realized that the visuals we were going for bore a resemblance to *Super Hexagon*[3]. *Super Hexagon* served as a good comparison game because it had simple, geometric visuals with an emphasis on color, and the core gameplay was able to be compelling despite being as simple as (if not more simple than) the game we were planning to make.

We did not find other games that made use of kaleidoscopes in the way we intended to, but we did find an “endless kaleidoscope” simulation with code provided online[4]. While we did not use the code in our project (it was written in the Processing 2 graphical library), it provided a good reference point for how to generate striking-looking kaleidoscope simulations from randomly-generated geometric shapes on a canvas.

---

[1] Citation: https://www.youtube.com/watch?v=q2fIWB8o-bs
[2] https://en.wikipedia.org/wiki/Canabalt
[3] https://superhexagon.com
[4] http://beautifulprogramming.com/endless-kaleidoscope/



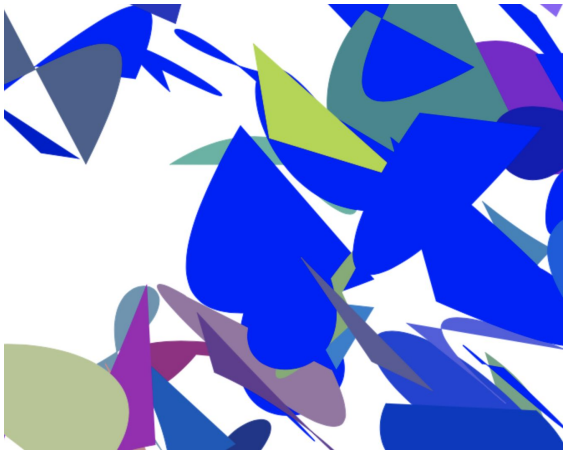
Methodology

The pieces that had to be implemented were:

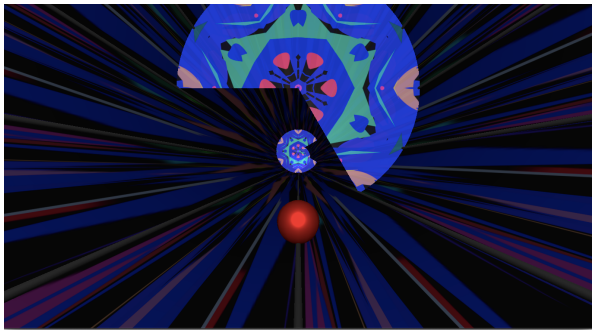
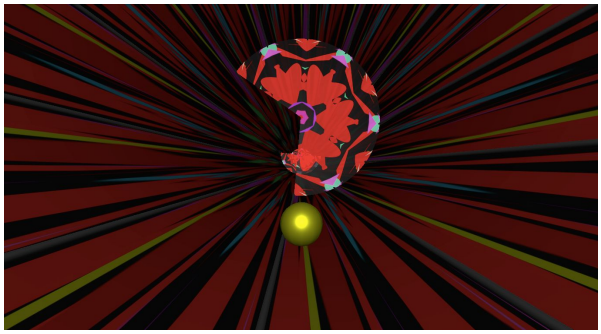
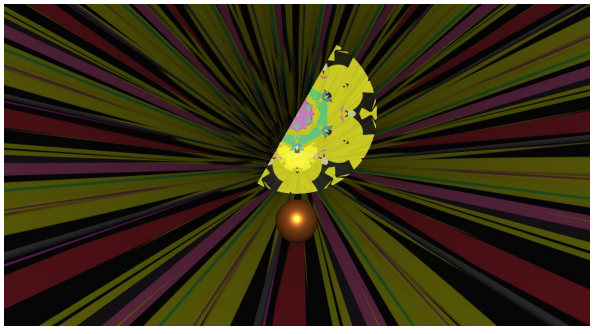
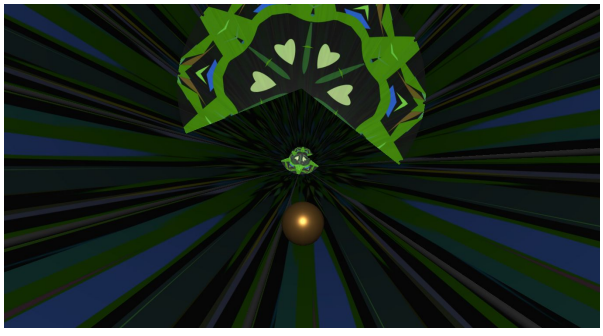
- 2D shape simulation
- Basic scene setup (controls, player, enclosing cylinder)
- Barriers (generation, geometry/kaleidoscope effect, collision detection)
- Player physics
- Gameplay miscellany (difficulty ramp-up, losing screen, practice mode)

2D shape simulation

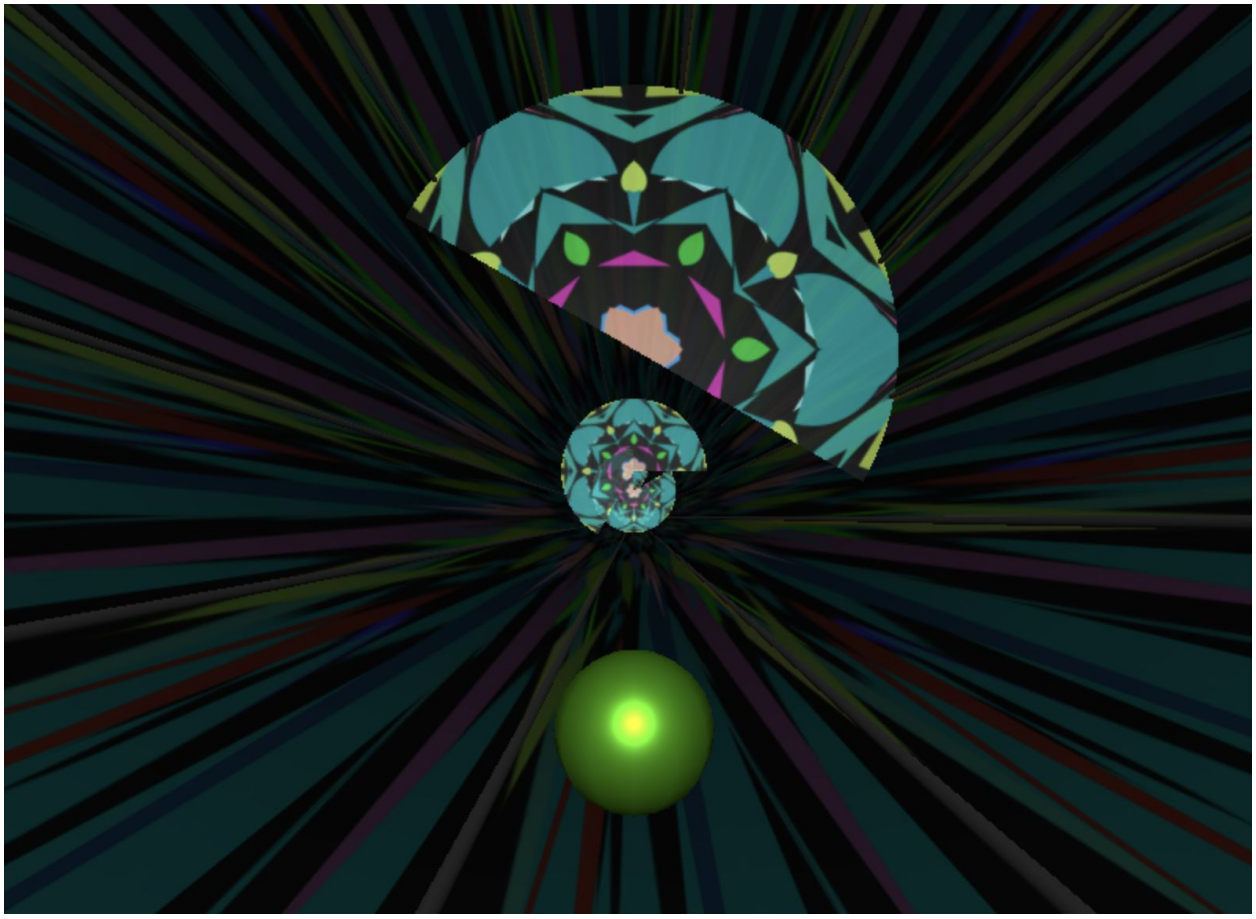
We created kaleidoscope patterns by procedurally generating shapes on an HTML5 canvas, and then mapping the canvas onto the in-game barriers using a CanvasTexture, described in more detail below. Each shape is initially drawn with a random position, direction and velocity, and an animation loop causes the shapes to travel across the canvas, resulting in kaleidoscopic visuals which constantly evolve throughout the game. Two categories of shapes are generated: triangles, and arbitrary shapes defined by Bézier curves between two random points (with randomly-generated control points as well). We tweaked the parameters, number of shapes, and canvas size until the kaleidoscope effect in the game was visually interesting and realistic.



The colors of shapes were initially randomly chosen and static, but we later decided to define the aesthetic of the game using “theme colors” which would change over time. To achieve this, each shape is constantly transitioning from its current color to a different random color by interpolating the R, G and B values of the two colors. In addition, the entire simulation has a theme color which is returned as the “random” color 60% of the time, and which changes approximately every 20 seconds. The color of the ball also changes whenever a barrier collision is detected, adding an extra visual element for players who play a single game session for multiple rounds. The screenshots below exhibit how the game’s color scheme changes as the theme color rotates.

Basic scene setup



Like many people in this class, we used ThreeJS heavily. The basic scene consists of the player (a sphere) rolling down an enclosing cylinder. The camera follows closely behind the sphere, and the player can use the arrow keys to roll left and right and the spacebar to jump. We implemented our own basic camera movement—the camera follows the player, where its speed is proportional to the square of its distance from the player. We found that this yielded very realistic-looking movement that was robust to all the changes we made to player movement over the course of development, and gave a nice sense of speed and motion during playtesting.

As the player progresses, kaleidoscopic barriers and enclosing cylinders are constantly being generated as the player approaches and destroyed as the player passes them. We made heavy use of THREE.CylinderGeometry for the enclosing cylinder (as an open-ended cylinder) as well as the barriers (as closed-ended ones). The theta.* parameters in the CylinderGeometry are particularly worth mentioning, as they allowed for a partial-cylinder geometry where only a theta-angle slice of the cylinder is constructed. As you'll see in the next section, this was crucial for implementing the kaleidoscope visual effects for both the enclosing cylinder and the barriers. The barriers and the enclosing cylinder have the same CanvasTexture applied to them; every

other generated enclosing cylinder is reflected across the x-y plane in order to ensure that the textures line up smoothly as the player moves down the tube.

Barriers

Each barrier is actually comprised of 2 to 12 pieces (referred to in the code as “pie slices”) implemented as CylinderGeometries with a CanvasTexture to apply the 2D kaleidoscope simulation as a texture. The mirroring effect of kaleidoscopes is achieved by instantiating all of the slice meshes with identical parameters, reflecting every other slice[5], and then rotating each slice around the world Z axis by a different angle in order to produce a continguous-looking barrier[6]. By producing multiple identical slices before applying transformations in order to position them correctly, we avoid needing to individually change the UV mappings of the CylinderGeometries created -- this process guarantees that the desired kaleidoscope effect will be correctly mirrored.

Implementing collision detection between the player ball and the barriers was particularly tricky. We had parametrized barrier creation according to number of “slices” and the angle of the center-line of the gap that the player was meant to go through. From this we were able to calculate the updated center-line angle based on how the world Z rotation had changed (from the player rolling left or right) since since the barrier’s creation. This allowed us to calculate a set of points around the circumference of the barrier such that, if a ray from the center axis of the cylinder to any of these points intersected the player ball, we had detected a collision with the barrier. After this was implemented, we simply checked in each timestep whether the player had collided with any barrier with a position.z close to that of the player.

Player physics

We iterated over several options for in-game physics, including trying to use the physics engine Physijs[7], but we quickly discovered that it does not work with open-ended cylinder meshes and is not actively being supported. Instead, we decided to implement our own physics between the ball and the enclosing cylinder. Implementing physics was a tradeoff between achieving realism, and making sure that the gameplay is fun, fluid and not overly difficult.

Since the ball’s forward motion (along the z-axis) is controlled by the game, we fortunately only needed to implement physics in the x-y plane. At each frame we create a new ThreeJS Vector3 (with z = 0) to accumulate forces acting on the player, and we use this vector to update the

---

[5] This caused unexpected challenges, as it entailed applying a transformation matrix that scaled the y-axis by a negative number, which complicates object-coordinate frames of reference in ThreeJS. The Object3D.rotateOnWorldAxis function helped a lot with this.

[6] See https://github.com/ewilden/collideoscope/blob/f8fdb34894db133283d7b144ff6f3aa6498c24a0/js/objects.js#L101.

[7] https://chandlerprall.github.io/Physijs/

player's current velocity and position. Forces on the player include gravity, a constant vector pointing in the negative y direction, and friction, which is based on the player's location relative to the bottom center of the cylinder. In addition, if one of the arrow keys are being pressed, we add a force to roll the ball in the appropriate direction along the cylinder. Finally, after we update the player's position, we clamp the position to the interior of the cylinder, which in effect simulates a normal force and keeps the player from falling through the ground.

To jump, we simply add another force to the player in the direction of the center of the cylinder. Initially, after implementing jumps, the physics felt very "sticky," and the player could stay on the ceiling or wall of the cylinder for 1 to 2 seconds before falling. We soon realized that after a jump, the player still had a component of their velocity being directed into the cylinder wall. This made the controls very predictable, and made the game almost trivially easy. When we took out this component entirely, however, the gameplay felt slippery and difficult to control, so we eventually we decided to retain some, but not all, of the player's radial velocity. We continued to adjust this and other physical parameters throughout development to make the gameplay feel natural and smooth.

Gameplay misc.

During normal gameplay, the ball gradually travels faster before capping out at a maximum speed. The speed increase is linear and reaches the maximum within around 30 seconds. This allows new players to get used to the game's controls and physics without dying immediately, but prevents experienced players from getting bored with long stretches of slow gameplay. There is also a practice mode in which the game does not end when a collision is detected, and player is allowed to adjust the ball's speed manually.

All of the 2D overlays over the game screen are implemented with HTML and CSS: the overlays start off with the property `display: none`, and when we wish to trigger an overlay (such as if the player pauses the game, loses, or enters practice mode), we use JavaScript to add a CSS class that changes `display` back to `block`.

Results and Discussion

We evaluated our end-product entirely through playtesting: we wanted to achieve a game with simple but compelling mechanics and appealing visuals, while adhering to the vision of "endless runner in a kaleidoscope."

We judge the project to be a success! Visually, while producing a *reflective*-looking kaleidoscope interior on the fly is difficult, we're quite pleased with the way the kaleidoscopic designs manifest themselves in the barriers and on the cylinder. We're also quite pleased with the physics-y aspect of the gameplay: there may be many endless runners, but there are relatively few where conservation of kinetic energy is a significant concern for avoiding barriers. While playing, we found ourselves learning strategies that we hadn't originally set out to include in the game, like rocking the ball from left to right to build up enough momentum to roll up the side of the tube. All in all, it seems like we have arrived at relatively unique gameplay despite the simplicity of design.

This project, like previous graphics assignments, tested our ability to visualize and reason about three-dimensional geometry in constructing the objects in the scene. However, because we were looking to achieve pleasing aesthetic effects (in terms of visuals and in terms of gameplay mechanics) rather than fulfill a specification, the project represented valuable practice at experimenting with and iterating on different ways to achieve a visual effect or an aspect of gameplay. Having to implement a more "real" collision detection and simple physics system as part of the gameplay was also very instructive; from a game design perspective, it was interesting to see where realism was aided by or at odds with playability, and our experience with tweaking the physics of the ball's interactions with the cylinder walls has given us new insight into how sensitive a game's movement can feel depending on small changes in physics calculations.

Conclusion

Collideoscope was intentionally designed to be simple in scope: we valued being able to produce a basic version quickly so that we would have the time to polish it into something we were proud to share and happy to play. Despite (or perhaps because of) this simplicity, we were surprised by how rewarding the time we put into the project felt. We hope any time our players put in feels equally rewarding!