SPongBall FlatPlanes aka A Well *Rounded* Spin on Pong

Victoria Zhong

1 Concept

Imagine the classic hit Pong, but instead of a flat paddle, it was curved and 3D where paddle can move along the y axis and the z axis. I'm still undecided whether the paddle should be a icosahedron or a perfect sphere. I think the icosahedron would be more interesting especially when things can bounce on the z axis as well. Angle of incidence will depend on the normal of the face it hits for the mesh sphere, and for the regular sphere the normal of where it hit and of course the walls. All in all there will be 4 walls encasing the "arena" the other two sides are the player areas where you move the paddle around and if the ball goes past points are scored. Scores will currently just print out in the terminal unless I have enough time to add text.

2 Deliverables

2.1 Paddle Movement

Controlling the paddles will be keybound to **w a s d** and **up down left right** for player 1 and player 2 respectively
The ball will be served from one person's paddle initially and move in one direction until it hits something. Note this serving will just be the ball automatically shooting out the paddle whereever the paddle may be, vs being able to move the paddle around to choose a place to serve ala brickbreaker.

2.2 Collision Detection

Program should properly detect collisions between ball and other objects such as paddles and walls. When collision occurs between paddle and ball or walls and ball, calculate the area of incidence based on the normal of the point that has been hit.

2.3 Textures

Add a texture to the ball or the paddles.

2.4 Powerups

Add powerups that ball could collide with that have various effects. I'm thinking of using bunny and bumpy cube as powerup items one to speed up the ball, and the bumpy cube is a caltrop that if hit gives the point to the person who did not cause the ball to hit the caltropl, and the ball disappears and is served by the person who scored.

2.5 Gaussian Blur

I'm not entirely sure how to make it fit in with the theming/aesthetics of the "game" but I could add gaussian blur perhaps as a "map" skin

3 Implementation

3.1 The Setup

The most important part of this is the setup. I created a 800 x 800 unresizable window. The bottom three tenths of the window so (-0.4 to -1) is dedicated to be where the future scoreboard and menu will be. The upper portion will be dedicated to the game area. There lies a rectancular cuboid that spans (-1, -0.4, +0.025) to (+1,+1,-1.975) in the screen positions from bottom left front to top right back. Inside this cuboid which will be referred to as the playing field and its surfaces its walls, there will be 2 spherical paddles and a ball. The paddles are of size 0.25 or 100 pixels. The ball is of size 0.1 or 40 pixels one twentieth of the screensize. The ball will randomly begin on one side of the screen and randomly move 0.05 (or 20 pixels) in 45 degree angle from the paddle in a combination of the x y or z direction eg it may move 20pixels 45 degrees towards us, or towards us and up. Periodically powerups will appear (with a 0.001 chance) up to 3 powerups on screen at most in a random location within the playing field. The bunny will speed up the ball and the caltrop (bumpy cube) will destroy the ball giving the point to whoever did not cause the ball to hit the caltrop. The cube will trap the ball for a few seconds and release it in a random direction All the randomness in the game will determined using the pseudorandom function rand which was seeded using the system time.

3.2 The Collisions

The technique used for detecting collisions is unfortunately not the most time efficient, but on a modern processor it should not effect the frame rate of the game itself too much. Because at any given time we can only hit one paddle, we only have to check one paddle and only if the ball is past two certain x positions which would be around ± 0.875 or 50 pixels from each edge. A better way to check is to check whether the distance between the barycenters of the paddle and the ball is less than the radius of the paddle plus the radius of the ball. We conveneintly have the barycenters stored in an array. (Note we plan to transition each of these "objects" to full Objects where barycenter, scale, and rotation are members of each object, currently they are all just separate parallel arrays)

Similarly we only test whether the ball collides with the cube if it is past a certain threshold, it can be argued that because I keep ball size consistant and do not have a powerup to shrink or grow the ball if the ball's barycenter is 0.05 or 20 pixels away from the playing field's walls it has definitely hit a wall and needs to rebound so we do that, also it would be easy to generalize it for all ball sizes. For the scoring walls where the paddles move around, the barycenter itself has to pass ± 1 because the paddle can still hit it back if the barycenter is still even slightly over and not touching the wall
There is an interesting special case where it hits both the wall and the paddle ie the corner shots and here you would just reverse the x direction and the direction of the plane it did not hit, if it hit both walls and paddles then it reverses completely completely because the ball would have been pinched in a way where it just goes back the way it came. If this is not how the physics would work, I can change it
Testing for collision with a powerup also uses bary centers of the powerup and the ball. Since each object is created to be scaled to one unit and then scaled to its final scale we can easily tell whether the ball is within another object's sphere.We don't need the preciseness of the full mesh. A bumpy cube actually is good as 2 cubes one rotated 45 degrees each way, but a circle will also do since the bumpy cube is so small that we can't really tell if it didn't hit the object completely or just clipped one of the points. Similarly we can use a circle hitbox for the regular cube representing the cube sucking the ball in. The bunny seems like a harder shape to deal with, however with the game and the way it is rendered, a lack of precision is not likely to be noticed, so we can use a sphere for the hitbox as well. A capsule can also work but it requires a capsuler rotated 45 degrees then rotated to the bunnys rotation again. However with a deadline very fast approaching, we will use a sphere for now.

3.2.1 Reflections

So after we detect a collision we have to handle reflections, for the cube it is simple we can just flip the directions on some axis, but it is different for the paddles.
The normal is actually the barycenter of the ball since regardless of where the paddle hits the ball, it will intersect the line between the barycenters of the two. It's a naive solution, but I can't think of a scenario where this is not true. After that, we use the classic reflection formula: $\vec{d'} = \vec{d} - 2(\vec{d} \cdot \vec{n})\vec{n}$ where d is the direction and n is the normal. Then we scale the vector to a length of 0.005. There is a bug where the ball can't escape from the paddle just move the paddle off the ball.

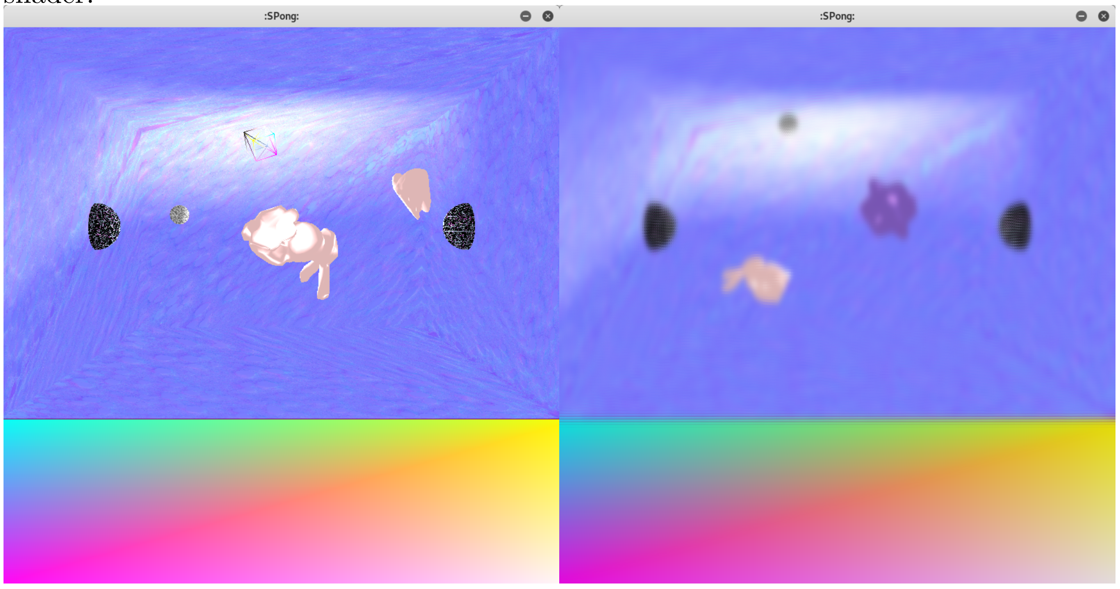
3.3 The Aesthetics

3.3.1 Textures

For textures we add a new VBO of UV's and also a variable to check whether the object will use a texture and what texture. I opted for each face being either the same pattern or alternating between adjacent patterns vs a whole model. I can't match the patterns in fabric when I sew, I won't be able to do it for the mesh. I chose to handle magnification and minification using GL_NEAREST. This creates a pixelated effect which goes well with the current aesthetic, also we plan on applying a blur filter after so we don't want it to be too blurry. You may ask why not use mipmaps, this is because I did make some mipmaps but noticed that you lose all texture for smaller samples and it becomes a gray square. I also opted for a more abstract pattern so I won't have to worry about shearing and skewing from the perspective factor. A normal map makes for a surprisingly nice icy texture.

3.3.2 Post Processing

I chose to do a gaussian filter. I changed the texture for the paddles because I think it would look better blurred, it is now this hideous grotesque stone like texture (that is made using a grid texture minified and repeated on each face). I had an issue where sometimes the program would stutter for a few frames like it was lagging very badly as well as the ball losing its texture when a powerup appeared, or just hang on a single framebuffer (I know this because I can see the wireframe of the quad overlayed over my scene. Also for some reason the first pass through didn't recognise the depth testing so there was some weirdness that needed to be fixed later. Also sometimes swapping between blurry and normal leads to crashes as well. It mostly did. But let's at least get the blur right. I grabbed the 9 weights from the open.gl tutorial. I personally prefer the learnopengl implementation because I can modify that to use the more efficient gaussian shader.


:SPong:
:SPong: