THE UNIVERSITY OF TEXAS AT AUSTIN, COMPUTER SCIENCE

Augmented Reality Brick Breaker (CS 354 Final)

Sean Kirmani and Eduardo Zapata

May 5, 2016

1 GOALS

We set out to build an augmented reality application because one of the fastest growing fields in tech right now is virtual and augmented reality. There ends up being a lot of blending between computer vision and computer graphics when working with augmented reality, but we wanted to focus more strongly on the computer graphics side. We decided to make an augmented reality version of brick breaker.

At the start, we want to be able to select and create two points in the real world, and then create a variably sized brick wall at those points. Then we would let tap the screen and they'd fire a ball, and the ball could collide with the wall and then be reflected along the normal of the collided brick, or the ball can miss the wall and we would lose the ball. While the ball is in play, the user is in control and is actually the paddle that the ball is bouncing off of. The player physically has to move in the real world to play the game.

We also wanted to make the bricks look realistic, so we wanted to give it some roughness in its texture. So we attached a **Perlin noise** texture to the bricks. Procedurally generating textures was one of the key goals of the project.

Another key goal of the project was the collision system. The ball needs to collide with the wall, and we also have the ball collide with the camera. The way to achieve this goal efficiently is to include a broad phase where we generate a **bounding volume hierarchy**, so that we can reduce collision detection from O(n) to O(n log(n)).

The final key point of the project was to implement a way to select points in 3D space. We needed to be able to select a point on the screen, and then use a plane fit to figure out the pose of the point on the screen. We not only needed to get the point in space from the depth sensor, but we also needed the normal at that point because we needed to set the orientation of the brick wall between the two points with spherical linear interpolation (slerp).

2 Implementation

We used Google's Project Tango and the SDK associated with it to implement the relative positioning which handles the pose estimation for us. We used the Rajawali rendering engine which is a wrapper for the OpenGL ES 2.0/3.0 framework on Android.

On application start up, we pre-generate some fixed amount of Perlin noise textures. We had originally implemented it where it would generate the textures for the bricks when creating the wall. It turns out that is much to slow to execute in real time. And it becomes very sluggish and slow. We did a lot of optimizations to generate the Perlin noise textures quickly. We end up creating small bitmaps and scaling them up with a filter to interpolate. We also apply a weighted opacity, and when we create the final texture, we blend this more and more opaque layers with the course base layer. This ends up being a slightly faster optimization to blending the pixels in a loop with an average, because it ends up just being an additive blending.

After the textures has been generated, the camera turns on, and we create a cursor to select the points where to generate the wall. One difficult challenge was determining how far the cursor was actually pointing. We used the depth sensor to determine how far in space the point exists. Our original idea was to use a 2D circle to select the point. A more intuitive way to interpret the depth of the point was to actual render a solid color sphere at that point in space. It is a much more intuitive way because as the depth sensor detects a point that is faster away, the sphere becomes smaller, and if it's closer it becomes larger. As we move around, the position of the depth sensor updates and we have to do a plane fit at every check.

There are limitations to to having a depth sensor based cursor however. The range of the depth sensor is optimal between 0.5 to 4 meters. So there are possibilities that we don't get valid depth data, and the cursor might not be set to a reliable location.

After the first point is generated, we generate a line from there to the new cursor location. When we select the second point, we add a brick wall, and then we send the line between the two points as the diagonal for the wall. We calculate the number of bricks in the x and y direction in the space of the diagonal's xy-plane and the orientation of the brick wall while generating the wall. Another challenge we found was that we needed to generate the bricks in the correct starting position because the second cursor point can be in 4 different coordinates depending on x and y being both positive and negative. When we generate the rectangular prisms for the brick, we also apply our pre-computed Perlin noise texture to the bricks.

Now that the wall is generated in the correct space, we can start playing. If the wall is created, then on a touch event we'll fire a ball is a ball is not already in play. The ball moves forward at some speed every frame. When we created the wall, we created a bounding volume hierarchy for the bricks to accelerate collisions. That generation is it's broad phase. Since the wall doesn't move, once it's set, our life is a little bit easier. We check collision in a relatively naive way. We simply check if the bounding box of the ball intersects with the bounding for of the brick. Sphere-rectangular prism collision could be more soundly implemented by using the radius from the center and checking if the bounding box is within that, but that ended up being a bit too complicate to implement for this project's timeline.

If the ball collides with the wall, we reflect the ball's orientation across the normal of the brick wall, which is identical to how we determine the orientation of reflection rays like in the ray tracer project. The ball can also collide with the camera. We do this check by making sure the distance between the ball is within some threshold of distance with the camera's position in world space. Unlike wall collisions, we don't reflect the orientation of the ball across the normal of the camera. We had originally implemented it this way, but this ends up feeling a lot less natural for the user because it ends up feeling that the user isn't really in control of the ball. We instead set the ball's orientation to be the orientation of the camera's look direction.

The final case is to remove the ball if there is no collision at all. We had to come up with some heuristic to determine if the ball in play is a dead ball. The way we determined that was by comparing the distance from the ball to the wall with the distance from the camera to the wall. The reason we used this heuristic is because if the ball's distance is larger and it's going towards the wall, then that means that the ball is behind the wall. If the ball has bounced off the wall, and the distance is larger, then that means that the ball is behind the tablet. An interesting mechanic of this implementation means, that the distance where the ball dies is variable. So theoretically, the user could run really fast to gain more time to hit the ball as long as they are moving faster than the ball.

And that's how we made an augmented reality version of brick breaker!