Portalesque

Ernest Richards
Polo Melendez
Jerome Zeito
Zach Moses
Portal-esque is a puzzle physics based side-scrolling game we developed over the span of 10 weeks. As the name implies, we based it off of the game Portal. The object of the game is to reach the end of the last level without being stopped by one of the many obstacles. Once passing the final level, you win. To get to the next level, there is a door you have to reach. The door takes you to the next level. You are equipped with a portal gun that lets you shoot two kinds of portals: one orange and one blue. These portals are connected so that if you go through one, you come out the other at an equal velocity. Not all walls are able to hold a portal; the ones that can all look the same while the ones that can’t look different and are easily distinguished. Different levels hold different obstacles. These obstacles can be lasers, spikes, moving platforms that can crush you, or even a deadly goat.

While making this game, we mostly used extreme programming. We started off with the Asteroids framework and moved from there, putting the most important pieces in first. We got in as much as we could by the end of the quarter using this method.

- Zach Moses
Design

Our design and implementation phases were closely linked due to the Extreme Programming method we undertook for this project. After we implemented something, we would build upon it with more design features. We would therefore, add another design element on top of what we already implemented.

The first design phase was to get a movable object with jumping capability. With the help of the jumping box framework given in Lab1 we were able to get this. The object could move left and right as well as jump using the space bar key.

The second phase was to get character shape and add other objects. We also wanted to have some stopping conditions as to not allow the character to go off the screen. We accomplished this in phase two, by adding a square object, and limiting the mobility of the player to within the screen area.
Phase three was to add a movable gun. This gun had to allow for bullets to be shot both on the floor, ceiling, and anywhere in between. We, therefore, needed to create a 180 degree rotate-able feature. We got the idea from the asteroids framework in Lab3. In this framework the ship, which shot bullets from the front, or tip, had 360 degree rotation. We essentially designed our gun to be like this ship and we were going to put it right over our player object to give similar rotation and bullet trajectory. We did, however, limit our rotation from the 360 degrees of the asteroids ship, to the 180 degrees needed for our purposes. In this phase we added a rectangle object to our player, and implemented the 180 degree rotation.

For phase 4 we needed to create a game world, a way to create levels and allow the player to interact with other objects in it. The Box2d physics engine allowed us to do this. It made the creation of the levels much easier, and also allowed collision detection between the player the world and other objects. It gave us the ability to easily create and add objects into the world. Implementation of Box2d is explained more in depth later.
In the screen shots above a stair object was created, a moving platform was introduced, and our player gained the ability to shoot horizontal bullets.

Phase five put the previous phases together, and also added the portal feature. By this time we had completed the fundamental design requirements for a side scrolling portal game. We incorporated the rotate-table gun with angled bullet trajectory. This was pivotal to portal creation by allowing much diversity into where portals could be placed. The gun was limited to two bullet colors, blue and red. The blue bullet creates the blue portal, and the red creates the red portal. If a portal is created with one of the bullets, and a bullet of that same color collides with another surface, the previous portal will disappear and a new portal will be created on the new surface. Later on we enabled portalable objects and surfaces to limit the areas in which portals could be created. This was crucial to the puzzle aspect of the game.

Portal transport was the most challenging and important aspect of this game. A contact listener was created, to allow the player to be transported from one portal to another. When contact between the player and the portal happened, the player would be ejected out from the other portal.

Phase six began the cleaning up of the game. We included image sprites, and backgrounds to our levels. We also enabled portal creation on sloped surfaces. Using the forest framework from our Lab3 assignment, we were able to embed 2d ppm images.
After we got a working framework for our game, we designed and implemented more objects to enhance game play. Without going into detail, objects created include: Buttons, Doors, Mines, Goats, Enemy bots, Cake, Elevators, Lasers, and the ability to implement much more.
Implementation

Using Box2d

A few weeks into the project we decided we wanted more realistic physics in our game. Since realistic physics are very difficult to make and optimize from scratch we picked up the Box2d library. Although Box2D has some downsides being used for a game that often needed to defy physics, it was still very helpful. It helped us organize our information, understand how to tackle problems we didn’t know how to begin, and unused variables sparked new ideas for more features to add to the game as time passed. Using Box2d revolves primarily around 3 different types: the b2World, the b2Body, and the b2Fixture. The world is where all of our objects reside. The world takes our b2Bodies and stores them in a linked list. The bodies stored this way will be impacted by the physics of Box2D. One huge advantage of using the b2World was memory allocation and deallocation. When working within the b2World the new and delete commands should never be used. Box2d allocates and deallocates memory internally for optimization. The programmer simply uses the Destroy and Create helper functions.

- B2World - Applies physics, allocates and deallocates memory, and stores the b2Bodies in a linked list, accessible using world->GetBodyList();
- B2Body – Primarily a way to link together fixtures, as well as apply certain variable values. B2Fixtures get stuck onto b2Bodies and are linked together by it. The angle, linear velocity, angular velocity, gravity scale, and other variables are accessed through the b2Body of your created objects. All of the fixtures attached to a body can be accessed with a linked list within the body, similarly to how bodies can be accessed through the world. Body->GetFixtureList();
- B2Fixture – This is what you see on the screen, and what interacts in your game. The b2Fixture is what has mass, collision, friction, density, size, and shape. The combination of these variables, among others, is what decides how your body will behave in the physics of your game.
- B2Joint – This is a less used object, but still very necessary in certain situations. A joint links multiple bodies in a certain way, depending on the joint type. For instance we used a Revolute Joint on the player and the gun to link those two bodies together. This allows the gun to rotate along a point on the player, as well as move along with the player as if it was being held.

Box2D was very useful for us, and helped us go much further with the game than we had originally thought possible. Looking at the way the files within the Box2D library and in the asteroids framework also helped us decide how we wanted to structure our source code. Using the library did not come without its drawbacks, however. The player and the portals defy physics inherently. This meant when dealing with these, which was often, we had to practically fight against the physics engine every step of the way. Player movement, jumping, momentum, friction, and density were all major issues. Portal shooting, managing, and travelling also proved to be the most difficult part of the game.

Algorithms

A variety of vector math was very useful in making some of our class objects more generic and reusable. By finding the proper equations needed, our moving platforms are much more versatile and
easy to use, as well as better collision and boundary checking with many other objects in the game. After implementing these new objects, we found that making a level in our game is actually pretty intuitive, though still time consuming with a lot of guessing and checking for coordinates and desired angles. Anyone with a little bit of guidance and imagination could customize their own basic level in our arenas.cpp. Here are some of the equations we found useful, and how they were utilized.

1. **Distance Formula and Direction Vectors** – This was used for the aforementioned moving platforms and boundary checking. Originally, or platforms all had a set horizontal path. We then upgraded them to be classes of their own, with their own left and right limit. This allowed us to change our horizontal paths to be longer or shorter based on our needs. Soon after, we decided that we wanted the platforms to be able to move diagonally, vertically, horizontally, and possibly even in a multistep path. For this, the distance formula was necessary. By giving the platform class a start and end variable that both held an x,y vector, we could set more versatile paths. Then, by checking the distance between the actual position of the platform and its destination (end) we were able to change the platforms direction, usually by swapping the start and end. This would cause behavior similar to inverting the Linear Velocity of the object. The relative order was as follows:

   1. Find the Direction as a Unit Vector: 
      
      \[
      \text{Direction as a Unit Vector:} \quad \frac{(end \ x,y \ vector) - (start \ x,y \ vector)}{(1/\text{Magnitude})}
      \]

   2. Check the Distance: 
      
      \[
      \text{Check the Distance:} \quad \sqrt{(position.x - end.x)^2 + (pos.y - end.y)^2}
      \]

   3. If Distance is < Threshold, swap start and end. Recalculate new direction. Else maintain current Direction.

   4. Use Direction and Speed to set the Linear Velocity: 
      
      \[
      \text{platform.SetLinearVelocity( Direction * Speed )}
      \]

2. **Ray Casting, Normals, and Reflecting**– As the game started to come together a request was frequently made by everyone we showed it to….. are you going to make lasers? To make the
answer yes, we needed ray casting. In Box2d all the fixtures used can cast a “ray”. A ray is basically a straight line that can be configured to return the things it collides with, and their normal. By setting the ray to return the information of the first fixture it hit and drawing over it, we had our laser. The next question was if we would make the lasers kill the player, reflect off of certain surfaces, and go through portals. The first of these proved relatively simple. All of the objects in our game were given user data. Our player for instance has “player” in its user data. So we simply set the ray cast to check if the object it was intersecting had player in its user data, if so kill player. For reflecting we used recursion, as well as for “travelling through portals.”

1. Ray Casting: Ray starts at point1. Point1 was usually equal to the coordinates of the object shooting the ray. Point 2 is where the ray ends. Point 2 was usually wherever the ray intersected its first object based off of the direction of the ray.

2. Recursive Ray Casting – Mirrors: By calling the function recursively, a new ray could be cast using the original ray for information. The new point1 became the intersection point of where the ray hit the mirror. The new point 2 was wherever the new ray hit its first object, based off of the mirror of the previous ray’s direction. To find the mirrored direction we used this formula: new point2 = old point2 – 2*NormalofMirror

3. Recursive Ray Casting – Portals: This was more difficult, but similar to mirrors. Instead of using the intersection point of the portal, we would use the coordinates of the second portal, causing the ray to hit the first portal and recursively create a new ray starting at the second portal. The ray never actually “passes through” the portal or “reflects”. The illusion of these two things happening is accomplished by shooting off new rays.

4. Using Rays for Portals: We eventually added ray casting from the player’s gun. Doing this allowed us to save the normal vector of whatever we shot a portal at, and apply it to the portal. This way our portals always had the same normal vector as the surfaces they were placed on. Using these normal vectors we were able to have more realistic movement through portals, as well as momentum when flying through.