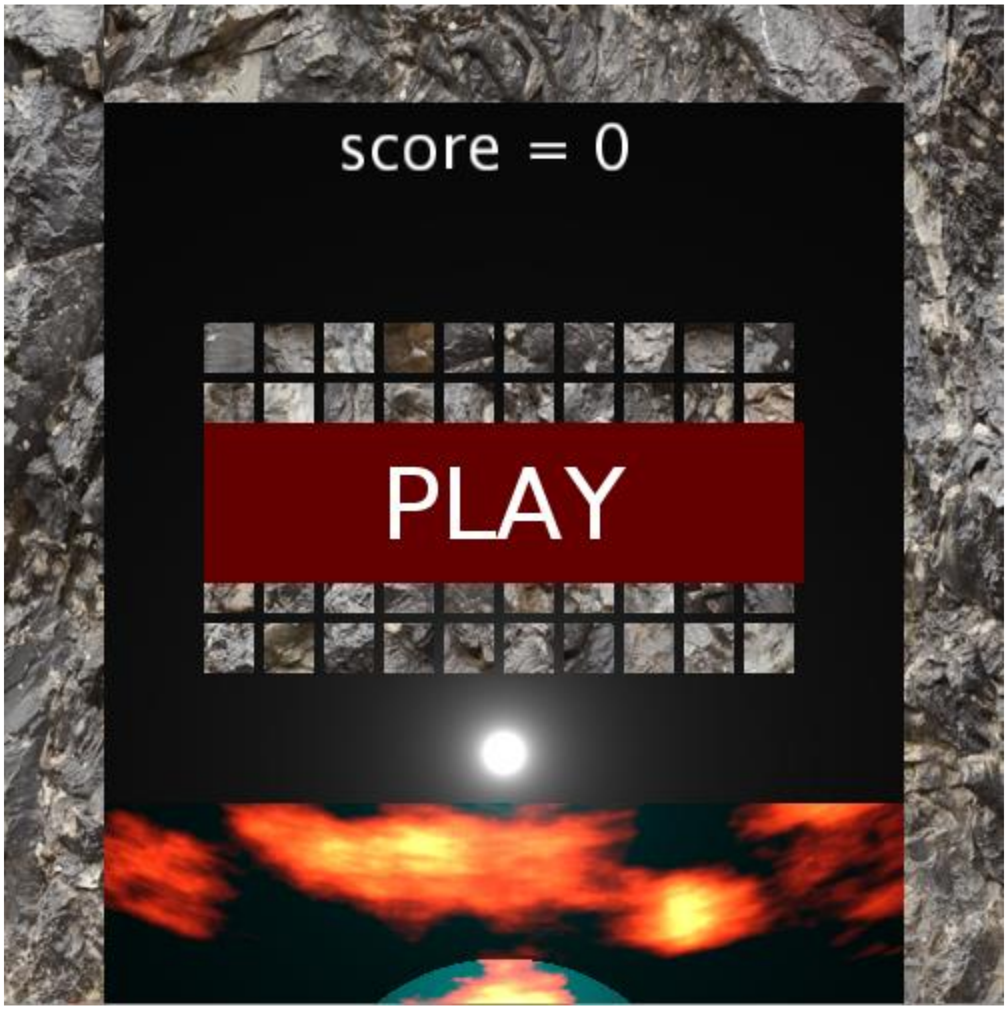
# Project 4 - Breakout

## Overview

For this project, you're going to implement a simple version of the game breakout. The core concept in this assignment is an introduction to collisions and movement. The background was hashed together from random glslsandbox.com demos.

## Description

Your creation might be somewhat similar to the following when you're done:



The game will have 3 main parts:

- Movement: Ball movement via velocity, and paddle movement via **keyPressed()**
- Collisions: circle-box, and circle-circle
- Background: using a glsl demo (don't worry: you don't have to write your own)

## Class Structure

You should create 4 classes for this project:

- Ball
- Paddle
- Box
- GameState

As before, each class should know how to draw itself and update its state. The Ball is the most sophisticated of these classes as it needs to know how to collide with other objects. The Paddle is the next most complex as you'll be need to update movement and draw an arc that's offset by just the right amount. The Box is pretty easy to implement as it's really just carrying information to draw itself and keep track of whether or not it's alive or a wall.

Because some of these functions are a tad tricky, you're provided with some boilerplate which includes comments. This boilerplate is simply a suggestion should you get stuck and need additional help.

### Class: Box

Let's start with the Box. You'll be keeping track of the x and y position of the top left corner along with the width and height so that it can be drawn. To draw a box, make sure it is alive, and then simply use the rect command with these class variables. It's alive by default and walls can't be killed. Rather than simply drawing a rect, crop part of a texture (perhaps via textures.com) or draw it through a shader.

### Class: Paddle

The paddle is a circle, rather than a box, so that we can more clearly aim. The trickiest thing about this is drawing the arc which uses the processing function **arc()**. The first two parameters are the x and y center of the circle, the next two are the major and minor axis (of the ellipse), the next two are the starting and ending angle (start < stop), and the last allows us to make it a filled in convex shape with CHORD.

Many game demos have a common movement bug which is that it performs an update when a keydown state is triggered. This is problematic because pressing a key which is intended to make a movement yields an update, a pause, then more updates at the speed that the key event is registered. The delay is undesirable, but it also allows the player to cheat by moving faster by changing the speed of a key event. For these reasons, we'll set a movement flag boolean variable to true when a **keyPressed()** event is triggered for our key of interest and we'll set it to false upon **keyReleased()**. This way, whenever update is called we will check the flag and move correspondingly. You can observe the bug by setting the frame rate to 1 with **frameRate(1)** and then hitting the key a whole bunch between frames and then compare that to the update with the flag approach using a different key.

## Class: Ball

The ball has a position, velocity, diameter, speed, and number of kills. The update function simply adds the velocity to the position. The draw function simply draws an ellipse with equal minor and major axes (equal to the diameter).

1. Circle-Circle collision: Check that the distance between the centers is less than the sum of the radii. Then, calculate the reflection vector via R = V - 2(dot(V,N))N. N is obtained from subtracting the centers from each other and then normalizing. V is simply the incoming velocity prior to the collision. R is the direction the circle glances off of the other. Multiply R by your ball's speed and you have your new velocity.
2. Circle-Box collision: If the box is alive, determine where the new position would be and use that as the the circle's position in the equation at the bottom of this post https://yal.cc/rectangle-circle-intersection-test/ to obtain a boolean value for whether or not it's colliding. The reason to use the new position in this check is to prevent the ball from sticking when it collides. If the ball collides with the box you'll update the x and/or y velocity by simply multiplying that component by -1. If the box isn't a wall, set the wall to dead and increase the ball's kill counter. If the ball hits in the corner, treat it like a circle-circle collision.

## Class: GameState

The GameState object, created through main will hold the variables pertaining to the details playing the game (ball, paddle, boxes, shader, texture). The gamestate has an update function where various objects positions, collisions, and drawing are performed. For drawing the shader background, and ball as a shader, refer to the SDF_demo where a minimal example of shaders is demonstrated.

## Main

In the main.pde file, as in the other file, functions with suggested APIs have been provided for you. This file creates the game object and has a boolean for whether or not the game should update and draw or just draw. You'll toggle this boolean using the 'p' key (for pause). Additionally, you'll use ControlP5 to create a "Play" button that toggles this boolean. If at any time the ball runs off of the screen, reset the game state (or call init).

## Tips

1. It may be useful to place objects in the direction of the velocity to test collision.
2. It may be useful to have separate testing scenarios where you make several of these object and several balls so you can visually and thoroughly test your collision logic.
3. You can somewhat prevent collision issues checking if moving objects are about to collide based off of what their next position will be and then not updating to that new position.

## Background: Shaders

Shaders are responsible for almost all of the cool effects you see in movies and games. In processing we have the option of dealing with both vertex and fragment shaders but other types of shaders exist (geometry, tessellation, and compute). Vertex shaders “move vertices” of some geometry and pass the output to the fragment shader (also known as the pixel shader) which adds color and lighting to that geometry. Using the fragment shader for the full screen allows for neat post-processing effects like dithering, color correction, blur, lens, distortion, and many more.

To create a shader in processing name a variable with the type **PShader** and then use **loadShader(“shader.glsl”)** to prepare it. Many of the shaders on glsl sandbox use the uniform variables **resolution**, **mouse**, and **time** to pass information from the main program to the shader. To set a uniform variable use the **PShader.set** as in **mypshader.set(“time”, t)**. To help you get started there’s some shader code included to help you get started. You should be able to load any of the shaders from glslsandbox.com with little to no trouble. Some shaders won’t work because they use a backbuffer or other special features. It might help to remove lines starting with # like **#ifdef GL_ES**.

## Fragment Shaders

The goal of this part will be to rehash various demos (of your choosing) to make a cool background and way of rendering the ball. Try switching a demo shader’s mouse location with the ball’s position. Try combining effects from different demos. To complete this section have a background that moves via some shader code and somehow depends on the ball’s position. You can grab relevant code from glslsandbox and shadertoy as is provided in the shader demo code provided.

Now that you know what they do and how to use them, let’s see how they work. There are a few new data types: **vec2**, **vec3**, **vec4**. Shaders have a vec4(x,y,z,1/w) builtin variable that is the position(x,y,z) and scaling(w) of that particular pixel **gl_FragCoord**. The most important builtin variable is **gl_FragColor** which colors that pixel via rgba using values between 0 and 1. Using this information you can set the color to white with **gl_FragColor = vec4(1.0)**. Very easily a gradient can be made by replacing **1.0** with **gl_FragCoord.x**. In processing, gl_FragCoord corresponds to the **size()** function in **setup()** which means we really need to divide this pixel's x location by the x component of the resolution to constrain the value to [0,1].

Signed distance functions are math formulas which allow us to determine the distance from a pixel to a geometric construct and use the result to color the scene. A demo using the functions from link 2 is included in the project files showing the creation of a circle, line, square, and box. You're not required to understand the shaders to create your background, this is simply an introduction so that a curious person might explore.

### Shaders Links (for fun)

1. https://thebookofshaders.com/
2. http://iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm
3. http://glslsandbox.com/
4. https://www.shadertoy.com/
5. https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)

## Grading

| Item | Description | Possible Points |
|---|---|---|
| Circle-Circle collision | Correctly collide and reflect the ball using the given formula. | 20+10 |
| Circle-Box collision | Use the formula in the link provided to check for collision. | 20+10 |
| Move/Draw Ball and Paddle | Update the ball and paddle positions appropriately. | 5+5 (extra 5 if done with shader) |
| Border / Boxes | Build/draw walls to contain the ball and Boxes to hit. (can use texture) | 5+5 (extra 5 for using crop vs rect) |
| Extra | Experiment with an extra game feature: levels, screen shake, particle effects, etc. More than 10 points can be earned here for non trivial features. | 10-20 (points given for making it clear what you did and describing it with your submission) |
| Shader as a background | Find a shader and insert it as a background, be sure to provide a link to the original source. | 10 |
| Pause feature | 'P' pauses and unpauses | 10 |
|  | Total capped at 100 | 120 |

## Note

The collisions should be decent, they don’t have to be perfect, if you have issues when the paddle moves causing the ball to get stuck that’s ok.  However, make sure everything works if you have the objects stationary.