Breakout - Reflection

This paper will begin by providing some basic background information on the original Breakout game itself. It will then survey the members of our team, elaborating on overall strengths and limitations of each member. From there, the paper will explain some of the rudimentary features included in a standard version of breakout, and then discuss some of the more unique additions we made to the original game. We will explain how implementing these features forced us to expand our horizons within software development, and finally, conclude by detailing potential attributes that could be added in the future.

To begin, some background on the game. Breakout was originally created by Steve Wozniak in 1972. Influenced by the Atari arcade game Pong, it is conceptually relatively simple. The game can be reduced to three basic components: a ball, a paddle, and 'n' layers of bricks at the top of the screen. The objective of the game is to bounce the ball off the paddle and destroy a path of bricks which will lead to the top of the screen, once the ball has reached the top of the screen, the level is won.

Team

Our team, consisting of Sub Raizada, Kevin Tinsley, and Alex Moriarty, have created a Breakout-like game. For the most part, gameplay is similar among ours and the original, however, there are some discrepancies. Nonetheless, despite having an exact idea of what the final product should be able to do, this was no small task. Luckily, one of our of team members, Sub, had actually created a few games in the past, and was able to guide our team to success. The value of his experience was immediately recognizable after he constructed the *game engine*, a cleverly designed template, which we were able to use throughout the rest of the project to keep our code logically organized, easier to edit, and accessible. The engine is roughly based off of the one in libGDX. The game is divided into screens, with each screen responsible for only its activity. For example, no game logic code goes into the loading screen. The screens are managed through the *ScreenManager* class. There are a few modules outside the screens system, these include: *GameConstants, Assets,* and *Graphics* (including *Camera*). These store information and helper functions utilized by the other modules. While it sounds easy enough to understand, actually working within the engine, proved to be a learning curve for less-experienced group members. Despite this, by the latter portion of the project, we grew to appreciate this, new, increasingly familiar, object-oriented approach to programming.

Features

The game engine was designed to allow easy implementation of planned features, such as resizable windows, toggling between fullscreen and windowed, animations, gravity effects, etc. Basic features (from milestones one and two) included screens for the main menu, a break screen between levels, a pause screen (technically implemented in milestone 3, but the work to add it was minimal as the framework was already in place), highscore entry and display screens, and an instructions screen. In the game itself, the paddle could be controlled via the mouse or keyboard. The ball was subject to gravity, and its bounce angle changed based on where it hit the paddle. Brick graphics changed as their HP decreased (from a solid brick to a cracked brick, etc.), and an unbreakable brick was also implemented. Multiple brick generation modes were added over time, starting from an empty screen with no bricks, to random bricks placed randomly, then random bricks placed into a grid, and finally, levels that are loaded from a file. This setting, and many, many other settings are stored in the GameConstants class, making it easy to change the behaviour of the game. Other such options in GameConstants are debugging features, such as print FPS, run a profiler, set whether the game should grab the mouse, graphical features, such as toggling motion blur and animation effects, and changing game variables such as speeds and HP and sizes of various elements.

List of standard features:

- Bricks loaded from files that specify the layout of each level (8 playable levels, and one more displayed in the background of the main menu)
- Paddle moveable with mouse or keyboard
- Ball with initial random direction and gravity, and change in angle of reflection based on where it hits the paddle
    - A random initial position was used, but this was later clamped to one specific position to make the game more deterministic, as having the ball start lower would result in making it harder to get to the top.
- Bricks with different HP values
- A scoring function which takes into account time elapsed and bricks destroyed
    - Scoring is nonlinear, proportional to 1/time, and then multiplied by percentage of bricks remaining (as measured by brick score, not actual number of bricks). Each level has a par time which can be adjusted to normalize scores across levels.
- Current game statistics (score, lives, time, level) shown in-game
- Persistent highscore saving, with a 3-letter name, and a highscore viewer accessible from the main menu
- User must manually resume the game after the ball hits the bottom of the screen and is reset, or when initially starting the game
- Resizable and fullscreenable window

- Motion blur
- Highly and easily configurable game behaviour via the usage of GameConstants
    - Some of this configuration is automatic (but can be manually overridden)
    - For example, some features cause bugs or low FPS on macs, and are automatically disabled if the system is detected as being a mac
    - A manual ‘low-performance’ toggle will disable graphical effects to significantly increase performance
- A loading screen which appears while assets are being loaded

Beyond the Standard Features

The milestone 3 feature extension included powerup bricks, which could add an extra ball to the game or clear a whole row of bricks, and added polish to the game, such as a functional and good-looking button system (buttons change appearance when hovered over with the mouse), a game automatically playing in the background of the main menu, and having the game darkened but visible in the background of the pause screen. Furthermore, we added lots of animations (smoke effects when the ball hits the walls, electricity surging through the paddle on collision with the ball, dust falling from bricks that get cracked, destroyed bricks breaking apart into pieces, explosions when a ball hits the bottom of the screen), updates to graphics assets (more detailed images/textures), and screenshake, which significantly improves the feel of the game when used with animations and motion blur.

List of extended features:

- Screenshots can be taken in-game
- Main menu screen has a game playing in the background
- Game is visible in background of pause screen
- Bricks that have special effects
- An elegant and beautiful user interface (in menus)
- Amazing graphics effects

* The features were combined into the following categories to fit the requirement of adding three new features:

1) Graphics effects - animations, screenshake, enhanced textures
2) Prettification of game UI - beautiful buttons, text, & menu design; game in main menu and behind pause screen
3) Brick powerups

Software Development Process (some redundancy with the features section)

As a team member had prior experience with creating video games, the breakout engine was designed to be very flexible and able to accommodate anticipated requirements and features, while retaining simplicity and good object-oriented design. It’s core was based off of the Screen and ScreenManager class. The ScreenManager class implements the main game loop, which calls the current screen's update() method at 60 times per second. ScreenManager also has a method to switch to a new screen. The Screen class contains just the update() method, as well as a frame counter (starts at 0 and increments each frame, used for animations). Since every screen except the actual game screen has clickable buttons, the Screen class also contains a list of buttons, as well as methods to draw all buttons, activate buttons given the position of a pygame mouse click event, and a empty method (like update()) that would be overridden by subclasses to handle button clicks. The Screen.update() method takes care of basic universal needs: handling pygame quit and window resize events, taking screenshots, and incrementing the frame counter.

Aside from these two core classes, there were four other significant classes: Assets, which loaded and stored all assets (such as images and animations, and perhaps sounds if that were to ever be implemented); GameConstants, which holds constant values that are universal to the game; and Graphics. Again, the game draws onto a fixed 1920x1080 surface, which is scaled to fit onto the window. The Graphics module manages the display (such as resizing and scaling the window, toggling between windowed/fullscreen, clearing the screen, flipping the game screen to the computer monitor, and converting mouse positions from screen coordinates to game coordinates). Finally, the Camera class, while stored inside the Graphics module, is an independent class that helps implement screenshake. It has a method to enable screenshake, called when the main menu screen is loaded (if screenshake is enabled in GameConstants), and a method kick(int), which increases screenshake. The Graphics module uses the Camera class to shift its rendering.

The parts of the game itself were implemented using something close to an entity-component-system pattern. The game/gameClasses folder contains various components, such as position (variations for a point, rectangle, or circle), velocity, acceleration, rotation, and coloration (used to color animations in-game). The game objects are simply classes that are a composition of these components. For example, the Ball object has a PosCircle, Velocity, Acceleration, and nothing else. There is one more class which glues everything together - Blittable. This represents a generic object that can be drawn to screen, and all things which appear in the game screen extend Blittable. Blittable defines a self.image, which stores either a pyagme.Surface (a static image), or an instance of the Animation class. To abstract away this complication, it provides a getImage(frame) method, which takes in the current frame (stored in the screen class), and automatically returns either self.image if self.image is a Surface, or self.image.getFrame(frame), which returns the current frame of the animation, if self.image is an Animation. Thus, rendering the game is as simple as looping over all objects, and calling Graphics.surface.blit(object.getImage(frame), object.pos), assuming pos is a PosPoint or one of its subclasses (rectangle or circle).

The game logic is roughly separated using a model-view-controller pattern. The GameState class holds only data - game objects such as ball and paddle, which are composed of only the components, such as Velocity, PosRect, etc. There is also a LevelTools class used to generate GameStates for a new level. The GameRenderer class has a method which takes in a game state, loops over all of its objects, and draws each object's getImage() at its position. Finally, the GameController class takes a GameState and performs all game logic - while GameState is purely a data storage container, GameController only manipulates the data.

The basic screen system proceeds in this manner once the game is started:

- LoadingScreen
    - Draws the 'loading' image, loads assets, then immediately switches to MainMenuScreen
- MainMenuScreen
    - Quit, which exits the game
    - HighscoresDisplayScreen or InstructionsScreen, which both return to main menu
- NewGameLoaderScreen
    - Uses LevelTools to generate a GameState, then initializes a new GameScreen
- GameScreen
        - Ties together game state/controller/renderer into one place
        - While GameController deals with game logic (manipulating elements inside the game world), GameScreen deals with external events affecting the game - pausing, having to continue once the game is over, etc.
        - Can go to PauseScreen, which returns back to this GameScreen or goes to a MainMenuScreen
    - BetweenLevelsScreen
        - Takes level number, score, and number of lives from GameScreen upon winning a level
        - Uses LevelTools to generate the next level, switches back to a GameScreen when the 'Continue' button is pressed, or goes to the MainMenuScreen if 'Quit' is pressed.
        - The cycle of GameScreen -> BetweenLevelsScreen -> GameScreen continues until a level is lost, or all levels are complete. Then, the game advances to one of the following:
    - HighscoreDisplayScreen
    - HighscoreEntryScreen
        - Takes a name, then adds score to highscores and switches to HighscoreDisplayScreen
- Highscore screen --> back to Main Menu

Given the highly organized nature of the code (which inevitably started to break down slightly toward the end of the project), adding new features was very simple. Adding the ball to the game is a good example. Once the screen framework was in place, the PosPoint/Rect/Circle, Velocity, and Acceleration classes were added. This was trivial, as they simply stored a few floats - there was no significant code written, with velocity.apply(pos) being the most complicated thing to do: it took in a pos and did pos.x += self.dx, pos.y += self.dy. Then, a Ball class was made with the following code:

```
class Ball(Blittable):
    def __init__(pos, velocity, acceleration):
        super().init(Assets.I_BALL) # load the image into Blittable
```

```
                                    self.pos = pos
                                    self.velocity = velocity
                                    self.acceleration = acceleration
```
The line

```
        Graphics.surface.draw(state.ball.getImage(frame), (state.ball.pos.x, state.ball.pos.y))
```

was added to GameRenderer. Finally, in GameController's update() method, we put:

```
        state.ball.acceleration.apply(state.ball.velocity) # dx += ddx, dy += ddy
        state.ball.velocity.apply(state.ball.position) # x += dx, y += dy
```

And that code concludes adding a ball to the game, with fully functional motion and rendering (no code needs to be changed to swap from a static image to an animated ball, other than loading an animation instead of an image into the Assets class). The only thing missing is collision detection. The Acceleration is defined in GameConstants as gravity, and the initial velocity is randomly generated in LevelTools when calling Ball's __init__() method.

An even more strikingly beautiful example comes with adding a game into the background of the main menu. As the game was completely encapsulated into the GameState, Controller, and Renderer classes, adding the game took only four extra lines of code, highlighted in red:

```
class MainMenuScreen(Screen):
        def __init__(self):
                super().init()
                self.gameState = LevelTools.makeState(99, 0, 1) # level 99 = main menu level,
                                                                 # 0 starting score, 1 life
                self.gameController = GameController(self.gameState)

        def update(self):
                super().update()
                ...
                gameController.update()
                ...
                Graphics.clear()
                GameRenderer.render(self.gameState)
                Graphics.surface.blit(background, ...) # has transparency so game is visible
                self.drawButtons()
                Graphics.flip()
```

While the above code doesn't make the paddle track the ball and doesn't restart the game upon a win or loss, it does embed a fully functional game into the main menu. As the event loop was handled at the top of the MainMenuScreen.update() method, most events will not reach the embedded game. Adding pygame.event.clear() right before gameController.update() filters out the vast majority of events, but the best solution was to have GameController discard input if `not isInstance(ScreenManager.currentScreen, GameScreen)`.

While having an organized engine designed with our specific needs in mind helped, there were still some obstacles encountered when making the game. Firstly, as Alex and Kevin hadn't worked with such a complex engine before, it took some time to get used to Sub's design choices and naming conventions. For example, it's already slightly odd that a PosRect can extend a PosPoint, but what makes it even more confusing is that for the ball, which had `self.circle = PosCircle(...)`, movement was done via `ball.velocity.apply(ball.circle)`, but for the paddle, which had `self.rect = PosRect(...)`, movement was done via `paddle.velocity.apply(paddle.rect)`. Nowhere was there a `x.velocity.apply(x.pos)`, even though the apply() method operated on a PosPoint. Over time, we all became more experienced with Python and this engine, its qualities began to become apparent, but the early difficulty made it crucial to have good teamwork and communication. Most of the work on this project was done together as a group of three, so that reduced any challenges associated with working in a team.

As a side note regarding the game development, it was elected to create a set of font images (75x75 pixel images with a letter or number in them) and then use these to draw things such as the score, time, and highscore names onto the screen, instead of using the pygame.font library. While the font library probably does provide ways to get the dimensions of a certain string so that it can be placed properly, there's a certain simplicity afforded by having every character be a fixed GC_IMGFONT_SIZE pixels wide and tall. A for loop could be used to load the characters into the Assets class using setattr(), so it was not tedious to write 36 asset definition and load commands.

Future work

Paddle movement in the main menu's background game is mostly effective, but there is much room for improvement. There is a bug with the pause screen on macs. On macs, pressing escape to exit the pause screen is automatically disabled by GameConstants, and so one will have to click the 'resume' button with a mouse to resume the game. Finding the cause of this and fixing it would be a good use of future time. The game seems to run very slowly on macs (tested on the latest MacBook Pro, as well as on a MacBook Air). It runs smoothly at 60 FPS on a Linux machine, which is odd, given that all systems use Intel integrated graphics, and the MacBook Pro and Linux machine had comparable processors. Switching the Linux machine to use a discrete Nvidia GTX 1050M GPU instead of the Intel integrated graphics had almost no impact on framerate, so there appears to be a CPU bottleneck. Printing the FPS stats (enableable via GameConstants) revealed that there was approximately 3 ms to spare per frame (out of a total of 17 seconds per frame for 60 FPS). Enabling low performance mode in GameConstants (automatically performed on macs) increased this to almost 10 ms of time spent waiting per frame, which means the time spent performing computations decreased from about 14 ms to about 7 ms - a huge improvement. These measurements came from the Linux machine, with an Intel i7-7700HQ processor.

Making the window fullscreen or maximizing it seems to increase compute time by 2-4

ms. Using the profile option in GameConstants (GC_PROFILE, uses the cProfile module) reveals that the vast majority of computation time was spent in pygame.surface.blit() [the surface is software rendered, explaining the lack of performance gain when running on the more powerful graphics card]. In Graphics, the game surface is first scaled to the size of the window surface, then drawn centered so as to have black bars on the sides or top and bottom, in case the game and window aspect ratio do not match. An optimization was made here, but it made it nearly impossible to resize the game, and would cause a crash in fullscreen if the monitor aspect ratio was not 16:9, so this optimization was reverted, because it only granted a reduction of 1 or 2 ms of compute time per frame. Future time should be used to see what else can be done to improve performance, as the game runs extremely slowly on computers with slower processors and on macs (about 10-12 FPS on the MacBook Air, and about 17 on the MB Pro).
The ball occasionally goes through a brick instead of bouncing off. This is very rare. The ball was also once observed to go through about 4 or 5 bricks almost instantly. The cause of this is unknown, and fixing it would be a good thing to work on in the future.

Earlier on in development, there was a massive bug with the ball losing speed over the course of a level (on all systems), and also with the ball reaching a significantly lower maximum height on the macs than on the Linux machine. After the extra ball powerup bricks, any effect of this slowdown has been lessened, because newly spawned balls have the proper initial speed. Casual gameplay reveals that if this bug is still present, it has no real impact on the game. The bug regarding the ball's maximum height has not been tested with the final version of the game. Future time would be well spent finding whether these bugs still exist, and fixing them if they do.

Finally, there is one last notorious bug. The quit button (red X in top right of window; hasn't been observed with main menu's internal quit button) occasionally randomly doesn't work. This happened frequently with earlier versions of the game. Adding to the pygame.QUIT event listening code the statements `print("looking for quit events")` and `print("quit event found")` seemed to completely eliminate the bug. However, the bug reappeared about a week later, even with the print statements in place. The bug then disappeared around the end of milestone one. However, it resurfaced a few times one day shortly after milestone three was started, but has not been observed since. It is unknown whether this also happened on the mac computers, and whether it is present in the final version of the game. However, pressing the quit button again a few (1-15) times will make the program quit.

Code Listing

C200_Breakout_Team12.py

    GameConstants, Graphics, Camera, Assets

    Highscores

ScreenManager, Screen, Button

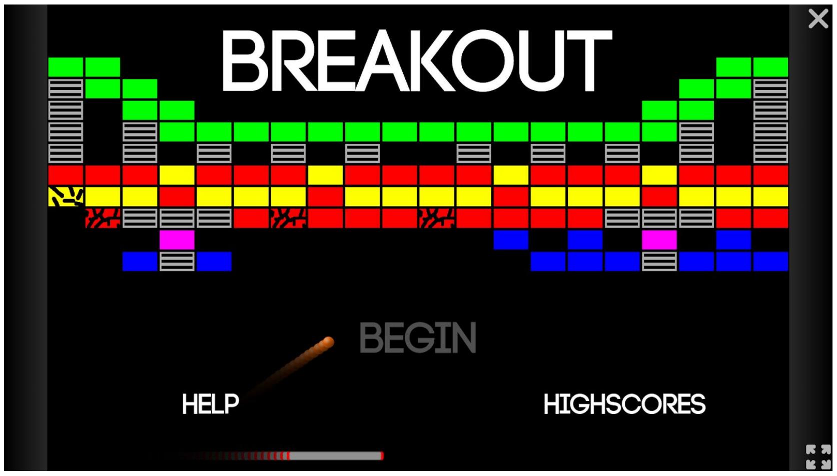
    LoadingScreen

MainMenuScreen
HighscoreDisplayScreen, InstructionsScreen
NewGameLoaderScreen
GameScreen
BetweenLevelsScreen
HighscoreEntryScreen

GameState, GameController, GameRenderer, LevelTools
Blittable, Animation
Game object components:
Position, Velocity, Acceleration, PosPoint, PosRect, PosCircle, Rotator
Game objects:
Ball, Paddle, Brick, Displayable (used for graphical effects, no effect on game logic)

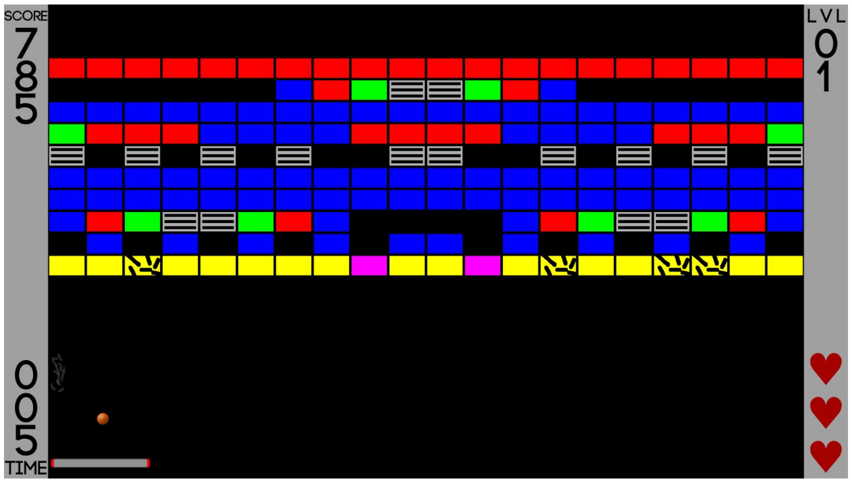
Images

Main Menu



Note that the 'BEGIN' button is highlighted, as the mouse is hovered over it (not visible in the screenshot). On the right are buttons to exit the game and to enter/exit fullscreen. The embedded level has all the effects and properties of a normal game. Motion blur is visible here. While very hard to tell by looking at the photo, the ball actually moves in a parabolic, not linear, path, due to gravity, and the blur trail does follow this.

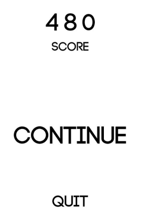
In-Game 1:



This is with low graphics mode enabled (default on mac, and manually enableable in GameConstants). Note the lack of motion blur. However, there is still dust remaining from when the ball hit the left wall. This effect can have different intensities based on how hard the ball hits the wall; the currently visible effect is a nearly-complete medium impact animation. The yellow bricks have cracked to 1 HP (max 2 HP).

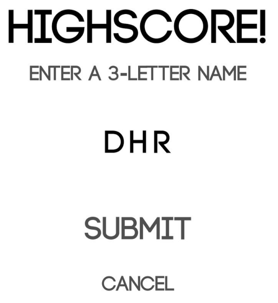
This picture shows many graphical effects. Screenshake and background flash are not visible. However, one can clearly see the (half-formed) explosion from an extra ball hitting the bottom of the screen. Also visible are fragments from a blue brick that was recently destroyed. These fragments are generated from a few white templates, which are programmatically coloured to match each brick and to fade out over time. Each fragment has a random velocity, acceleration, and rotation, change in rotation, and change in change in rotation. The green bricks show their 1 HP and 3 HP states (while a 2 HP state does exist, it is not visible here).. The yellow bricks spawn an extra ball when destroyed. Any one of the cracked (1 HP) purple bricks, if hit again, will clear their entire row. The gray bricks are invincible, and can only be destroyed by a purple brick's special effect. Aside from screenshake and background flash, also not visible are the dust particles that appear when a ball hits, but doesn't destroy, a brick. However, one can see the 'electric' effect that goes through the paddle when hit by a ball. There are multiple versions of this effect, for when the paddle is hit on the left, right, or center, or when hit very hard.

<u>Between Levels Screen</u>:



As with the main menu, buttons react to mousing over them. Shows your score and options to continue or quit the game.

Highscore Entry Screen:



HIGHSCORE!

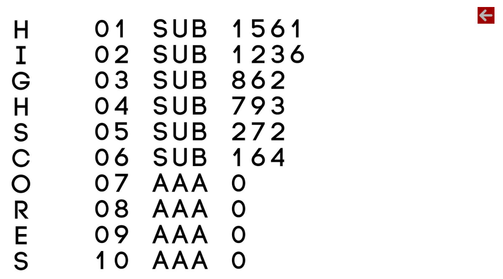
ENTER A 3-LETTER NAME

DHR

SUBMIT

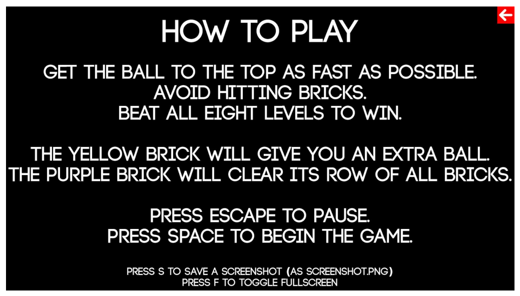
CANCEL

The submit button will have no effect unless the name is three letters. Backspace is handled correctly. Typing letters when three are already present will have no effect. Canceling will not submit the highscore, but still go to the highscore display screen:



H	01	SUB	1561
I	02	SUB	1236
G	03	SUB	862
H	04	SUB	793
S	05	SUB	272
C	06	SUB	164
O	07	AAA	0
R	08	AAA	0
E	09	AAA	0
S	10	AAA	0

The back button in the top right becomes a brighter shade of red and has a more pure white arrow when hovered over (every single button in the game has a hover effect). This is visible in the following photo of the instructions screen:



HOW TO PLAY

GET THE BALL TO THE TOP AS FAST AS POSSIBLE.
AVOID HITTING BRICKS.
BEAT ALL EIGHT LEVELS TO WIN.

THE YELLOW BRICK WILL GIVE YOU AN EXTRA BALL.
THE PURPLE BRICK WILL CLEAR ITS ROW OF ALL BRICKS.

PRESS ESCAPE TO PAUSE.
PRESS SPACE TO BEGIN THE GAME.

PRESS S TO SAVE A SCREENSHOT (AS SCREENSHOT.PNG)
PRESS F TO TOGGLE FULLSCREEN