

Python Brawl

Final Report

Submitted to

The Faculty of Operation Catapult LXXXV

Rose-Hulman Institute of Technology

Terre Haute, Indiana

By

Group 4

Philip Mathieu

Wheaton Warrenville South High School

Warrenville, Illinois

Matthew Mercer

St. Xavier High School

Oxford, Ohio

James Savage

Archbishop Hoban

Clinton, Ohio

June 30, 2009

Introduction

Our goal in Operation Catapult was to learn Python, a high level programming language, by writing a program of our choice. Given a long list of program ideas, we set our sights on a video game, inspired by a widely popular Wii title, *Super Smash Bros. Brawl*. The game was very complex, so we had to cut it down to a reasonable size. One that we could complete in less than three weeks.

The basic setup of *Brawl* consists of multiple characters engaged in melee combat, attempting to push each other off the map, removing one of their enemies lives. In the interest of simplicity, we took only these two elements as our base of gameplay. We wanted to do more, like the aspect scaling *Brawl* uses to show all characters, even as they move farther away from each other. But due to time and the technical limits of PyGame, the Python library we used to actually draw the game interface, we simplified and condensed our ideas to reasonable size.

Method

All our group members entered this project with prior programming experience. Even before the concept of Object Oriented Programming¹ was introduced, we began carving our game up into objects. Roughly put, our game is a hierarchy of objects. At the top is the Opmstdcert, or Object Person Movement System That Detects Collisioner Thingy. Originally created as a joke class, we realized how practical it would be and kept it, wacky name and all.

The principal behind Opmstdcert is simple. When an entity want to move it calls Opmstdcert's `try_move()` method as the argument of a conditional (ie print "can move" if `opmstdcert.try_move(x, y)` else "cannot move"). Since the Opmstdcert also handles all game entities, it can cross check the point to all entity bounds, and determine if a movement will defy game physics. But it can do much more than just check for open space. All movement checks also test any collision for type, say a character and missile. In such a case it would then remove the missile from the draw loop and from its entity list, and call the `character.take_damage()` method.

With an object container made we set out to make the actual objects. These were divided into two sections of two subsections, making four unique types of entities in total. The first type, non-moving entities, make up the platforms and scenery. These entities differ primarily in their interactions with moving entities. Platforms stop entity movement and are drawn below the moving entity layer, (although this cannot be observed). Scenery does not interact with moving entities, and is drawn above moving entities, masking based on its alpha.

¹ Object Oriented Programming consists of divided programs into objects, with each object containing values such as strings and integers, and methods which usually perform some action or return data. It is rising in popularity due to the modularity and logical division of code it offers.

Moving entities are broken into two grounds, characters and generic. Generic entities are the simple, multi-purpose entity that is created as weapons fire. It initiates with a starting coordinates and fixed x and y speed. Character respond to multiple prompts, both user input and environment effects like knockback.

While the Opmstdcert handles entity management and calling each entity's draw method, it does not control the main run loop, except for halting executing, which is done by Opmstdcert's handle method.

```
# The code for the main run loop
clock = pygame.time.Clock() # Create a PyGame Clock object that we'll use to get
a constant frame rate
events = pygame.event.get() # Get the list of all events that have occurred
since the last time we checked for events
while True: # Loop infinitely (it's okay, we'll exit when we want to)

    control.handle(events) # Handle our current events
    if not global_conf.pause:
        control.act() # Have our actors act
    screen.fill(color)
    control.draw() # Call our draw function
    pygame.display.flip()
    clock.tick(fps) # Tell the clock that we want it to try to use the frame
rate specified above
    events = pygame.event.get() # Get all the events since the last time we
checked for
```

Characters contain all features of an entity, such as speed, but also contain more values for missile count, damage, knockback² and weapon charge. They change direction and speed based on dynamic factors such as user input, via keyboard arrow keys, and environment properties, such as character knockback. Characters also have specialized actions, like the creation of missiles and energy orbs. Characters pass a request to Opmstdcert with starting coordinates for the attack. From there the entities pass into the main run loop's draw cycle. Characters only remember themselves, and once fired an attack will affect all characters equally. However it is unlikely to be hit by one's own attack as the movement constant is faster for attacks than characters.

```
# The headers from the character class
class Character:
    def __init__(self, name, imgs, controls, startx, starty,
facing_right, screen, control_thingy):
    def __get_img__(self):
    def act(self):
    def get_name(self):
    def get_rect(self):
    def draw(self):
    def handle(self, events):
```

² Knockback is a force opposite to the player's desired motion that is created as damage is taken. It is a key part of the game as the only way to kill a character is to push them off a platform and have them fall beyond the baseline, removing one life from their count. With each increase in damming knockback increases based on a knockback constant and damage count, making it harder to stay on platforms.

```
def take_damage(self, is_high, from_right):
```

By this point we had a fairly stable sketch of how the mechanics of our application would work, now came a decision point. There are two ways to get content into applications, static and dynamic. In a static model, content is hard coded into the application source, making upgrades and changes hard. It has the advantage that it is natively stored and does not require additional data accessors, but we were looking ahead and thought it would be to our advantage to use a dynamic model. In a dynamic model, content is stored outside the application, allowing for changing content without modifying the actually application code.

```
# This is all the code it took to get XML files into a Python format
# From there all that was left was cleanup
if(os.path.exists(self.path)):
    element = ElementTree.XML(open(self.path+'MapFile.xml', 'r').read())
    for option in element:
```

However this added yet another choice. Since all our data, maps, starting positions, character images and world configuration would need to be stored outside our application, we had to decide how to actually store the data. It would have to be done in such a way that an automated process could take in a file, parse it for meaningful information, and defrost it into something usable by python. In theory such a process should also check for and handle errors, however we did not have time to implement this. As a result we had several errors involving corrupt map files from stray characters. However; when written correctly, map files are stable and reliable.

As for the actual format of the file, we bounced between writing Python scripts which would be imported and read directly into Python and XML, a markup language with very loose standards allowing for custom data containers. Importing Python was quickly dismissed for security reasons, which we considered more out of habit than necessity. In an included script you allow not only for data imports but raw code execution, potentially allowing for malicious code to be inserted into the runtime. XML would require more work to get into native python, but it allowed for a safe way to store and retrieve data. XML also has the advantage of being easier to pick up. Since it is not a programming language, but a markup, with the sole purpose to store information, it does not have the confusing syntax of a programming language.

```
<!-- This is the Hard Map's XML file -->
<?xml version="1.0"?>
<root>
  <option key="PBMapName">Hard Map</option>
  <option key="PBMapDescription">Part of the demo collection</option>
  <option key="PBMapAuthor">James Savage</option>
  <option key="PBBackgroundImage">Background.png</option>
  <option key="PBBackgroundColor">0 0 0</option>
  <option key="PBCharacterRunningBox">135 160</option>
  <option key="PBBackgroundMusic">Background.mp3</option>
  <option key="PBPlatforms">
    <entity identifier="purple_rock" face="right">0 150</entity>
    <entity identifier="purple_rock" face="left">1155 150</entity>
    <entity identifier="stone_floating" face="right">0 500</entity>
    <entity identifier="stone_floating" face="left">1024 500</entity>
```

```

        <!-- Character will not be placed here -->
        <entity identifier="stone_floating" face="right">Q max</entity>
        <entity identifier="stone_floating" face="right">1Q24 max</entity>
        <entity identifier="ground" face="right">416 200</entity>
    </option>
    <option key="PBScenery">
    </option>
</root>

```

The XML allows a lot of data to be stored in a compact format. In the interest of quick code creation as much as possible was inferred from the supplied data. For each key/value pair there was a different interpretation method. For file paths, the directory the file was in was inferred from its purpose. Platform identifiers actually refer to image names, and commas were removed from (x, y) coordinates to allow the default action of the `str.split()` function. However easily the data could be moved to Python, we decided to go the extra step, making it easier to implement. The Map Parser object took the raw input and pulled out important values like Platforms and Scenery, putting them in separate lists from the rest of the options³.

Once we had a working build that would load information into memory and theoretically draw it, it was time to introduce actual images. This had some problems at first, partly due to differences between how James executed the python, directly via a terminal on a mac, and the rest of the team, via an IDE⁴ on windows. Also from SVN errors involving images not committed to the shared repository, and thus not shared across all builds. Once these errors were resolved the program flowed together very well. The game was playable and worked, but still needed polishing. From there there were only a few features left to add and a dozen or so bugs to fix.

Results

With any computer program, there are bugs. Within our Python Brawl, there were many errors that had to be fixed in our two and a half weeks here at Operation Catapult. Originally, the creation of maps was painful and worked after many hours of debugging but, as said above, we decided to define the maps in XML for succinct and simple map creation.

Following the XML creation, we quickly added characters to our maps foolhardily and encountered many errors all revolving around the same overlook in coding. Characters would appear in platforms, under platforms, and off screen. The solution was to alter the `Opmstdcert` so that characters could not start within the platforms.

Now that the characters could actually move, game play could actually begin but our next error came quite quickly. There was a glitch with the `Opmstdcert` class in which collisions between character and platforms were very irregular. Collisions from the side and bottom were

³ See fig 1 for more information on XML, python data and explanation of data contained in the Platform Object

⁴ Integrated Development Environment. A tool providing enhanced editing utilities for a particular language. In the case of eclipse, file management, code completion and inline documentation.

completely ignored so that the character would enter the platform and not be able to move after in entered the entity. While being stuck in a platform made the character invulnerable and would doubtlessly be an amazing tactic in game play, it was unrealistic and we discovered the error in our ways. Including side and bottom collisions in the Opmstdcert, the next error was in the characters being an odd number of pixels above the Platform entities. The error was not in the programming this time but in the sprites (images) of the character entities. Once the .png file was edited, this problem was solved.

A completely random error that we fixed in the coding involved extreme lag⁵ and was due to the way we displayed our characters. Our way was completely correct in its execution, but Michael Boland, student advisor to the CSSE Operation Catapult department, taught us a way to convert the .png file into a more native image for PyGame to display. Once this new method of displaying characters was implemented, the game ran smoother making our game seem like it would be fun to play.

All of the above errors were found and solved before we added character movement (before the characters just slid around the screen). But once there was movement that could possibly be considered as graphics, there was a glitch where the character disappeared every ten frames. The error was in the sprite sheet where the image that was displayed was completely clear, causing the character to disappear. Once corrected, it ran quite smoothly. In the end, we created a realistic game that modeled the core rules of “Super Smash Bros. Brawl.”

Analysis

Initially we had picked a D&D style RPG Adventure Quest based game. After thinking about the amount of code needed to have the game engine run the quest simulation, we decided to move to something simpler. This turned out to be a n overall good decision. Not only was the *Brawl* game easier to visualize and design, it allowed al members of the team to play together, was easier to test, and easier to puck up and play.

Looking back from our initial idea of a D&D Quest game we were very smart to think smaller. And while it seems like we chose a smaller task to program, there is also a sense of accomplishment. We did a lot in these two weeks with our game. While we all had prior programming experience, many had never made an involved game. Concepts like draw loops and action handlers were new and required some thinking. Our game didn’t reach all its feature requests. Missing are things like aspect scaling to include all characters in view, like in Brawl. This would be hard to do in python, as the main view drawing was done through drawing PNG images to scale by supplying (x, y) coordinates. To do true scaling would require drawing the game screen to an image, scaling and cropping the image, and then drawing that cropped region. We never actually tested this, but on top of its complexity, we worried about game speed and processing overhead required to do this.

⁵ Objects move slower than they are intended to do so, causing control of the Character to be difficult

Also missing are general tune-ups like pixel specific redraw, which would smartly figure out what regions need redrawing and only spend time updating specific areas. This was left out because of time constraints. We are quite happy with how our project turned out despite this. We got limited WiiMote functionality through a program called DarwiinRemote on the mac, which maps WiiMote buttons to keyboard keys. Initially we had intended to use a free library to access the WiiMote directly over bluetooth, but again were forced to ditch this in light of an easier solution because of time.

```
# To allow for quick keyboard configuration we put controller profiles in a list
controls = [[K_LEFT,K_RIGHT,K_UP,K_DOWN,K_SLASH],
            [K_a,K_d,K_w,K_s,K_e],
            [K_f,K_h,K_t,K_g,K_y],
            [K_j,K_l,K_i,K_k,K_o]]
```

Also purged from our release version is background sound. Although the code is sitting in the application, and the key exists in the map file markup, due to instabilities it never went through. However it is a feature we plan to try to implement in any free time. The game has a fairly reliable and customizable engine, capable of fully themeable maps, which in theory could be created by anyone. We will probably play around with it for a while, after Catapult has ended. But without any illusion, we realize the code is very bad quality since it was written in such short time with a single minded goal of just working. It would be interesting to see what would happen if the base engine was rewritten to be more flexible, and allow for more customizable gameplay. Specifically adaptations that would allow for a sidescroller or other one person game play modes.

Discussion

After finishing our project we all have decided we like Python, and it was worth the time to learn. After learning one language, you can easily pick up most others. This applies to all of us, who picked up Python about as soon as we realized how the syntax differed from the languages we used most. But the main interest Python draws comes from its high level⁶ nature. Our previous languages were mostly lower then python, such as Java and C, so writing in Python was quite fun, allowing us to focus more on what were were doing and less on the specifics of how. In languages such a C and Java, there requires many more lines of code to do the simplest of tasks. For example, simply outputting “Hello World” in Java takes at least three lines of code where as in Python it takes three words. Programming in this higher level caused programming this game to be really fun and allowed a lot of play testing. Operation Catapult was a success for us in our opinion.

⁶ A high level language is built on top of many layers of languages, and is usually simpler with easier to access functions. This in contracts with a low level language which interacts on a very basic level with components. The two both have their pros and cons. High level languages often sacrifice fine control for simplistic syntax and quick implementation, while low level languages take more detection to learn and more work to maintain and debug.



Fig 1