Improving User Engagement to Arcade Style Shooters through Color Matching

Reilly, Terrence Justice, James Goodall, Brad Maresh, Keefer

Gleason, Meagon

April 3, 2017

Abstract

Arcade style shooters enjoyed popularity throughout the 1980s and 1990s. Color and pattern matching games such as *Simon*, *Bop-It*, and *Twister* were also popular during this time. While the popularity of both genres may have waned in recent years, the purchasing power and free time of the children who grew up in these decades has increased dramatically. Few video games have tried to capitalize on the combined potential of these genres.

In this paper, we present a coherent schema for combining aspects of pattern-matching and arcade style shooters from the late 1980s and early 1990s. Arcade replicas, and color matching games from this era are plentiful, but most games which amalgamate the two concepts are based on rhythm and music (e.g. *Rock Band*, *Frets on Fire*, etc.) We further assess the balance of game difficulty necessary to maintain user engagement. We report qualitative results of players using a purpose-built color matching arcade style shooters built in Python. Research was completed using small focus groups.

Our results demonstrate that an engaging arcade-style experience is possible when combined with basic color matching. However, we present evidence suggesting that the action in such a game must be well-balanced to avoid becoming overly-difficult.

Background

One of the first electronic, microcomputer-controlled games, *Simon*, has also been one of the highest selling games of all time. *Simon* was invented in the 1970s by Ralph Baer and Howard J. Morrison, and was marketed through Hasbro. *Simon* greatly influenced later games, especially music and rhythm games such as *Dance Dance Revolution*. (**?**).

Part of *Simon*'s popularity came from its variable difficulty. Baer created *Simon* with a toggle switch allowing players to make the game more difficult as they progressed. *Simon*'s simplistic colors and low cost also were important factors in its success. Finally, part of *Simon*'s success also lay in the notion that gameplay could improve cognitive abilities. This belief is not entirely unfounded.

In *Cognitive Benefits of Computer Games for Older Adults*, E. Zelinski and R. Reyes discuss how studies have demonstrated that playing video games help with reducing or delaying cognitive diseases such as Alzheimers in adults. However, unlike *Simon*, the games which appear to have the largest affect in producing benefits are digital action games. The authors further point out that adaptive difficulty not only makes games more enjoyable, but also make patients more likely to continue playing them.

Many of the games from the era of *Simon* are approaching an age where they will be susceptible to cognitive decline. Althogh it may seem an intractable group for the gaming industry, 49% of gamers are 18-49 years old, with an average age of 35, and 26% of gamers are over 50 years of age (**?**).

In addition to the cognitive benefits to the older adults that these games provide, they also can provide support to a child's development. Dyslexia affects approximately 10% of children, but simply playing video games for a short period of time each day can help to improve a child's reading ability.

The study recognizes that children with attention dysfunctions tend to also have links to dyslexia, and it is known that attention dysfunction can be trained through video games. Within these games, only action video games helped to offset the effects, increasing reading speed and accuracy after twelve hours of playing. This increase in ability, in turn, would prepare the players for an easier ability to perform well academically as well as help players retain self-confidence that can be lost through learning disabilities. (**?**).

Video games help to prepare girls for a competitive future in STEM focuses on the role video games play in the higher-level spacial skill. Among the listed titles such as *Tetris* and *Pac-Man* is *Space Invaders*, a vertical arcade shooter.(**?**)

Leigh A. Hughes identifies a bias in the methodology in schools favoring sequential learning styles. A strong link is drawn between visual-spatial skill and excellence in STEM-related fields. By building confidence early on in the development of visual-spatial learners, video games can provide opportunities to more complex visual-spatial skill development, and in turn STEM careers.

Methodology

Overview of an Existing System

The game that we have drawn inspiration from is a 1980s arcade game known as Galaga. Galaga was developed by Namco in 1981 and is a shooter based game where the player tries to take down enemy ships by shooting at them. The enemy ships may be taken down in one or two hits. Enemy ships requiring two hits are known as the Galaga and will turn blue after the first hit, indicating that there is one hit left to take down the ship. While shooting at enemy ships the player must also avoid being hit by enemy fire while also avoiding the beam from the Galaga looking to capture a players ship. If captured the players ship will remain imprisoned unless the Galaga is taken down.

Galaga's gameplay consists of managing the movement and firing of a ship in order to avoid incoming projectiles whilst scoring hits on enemy ships by firing their own. The crux of this system's architecture as well as our own is the ship class. The fields for the ship class include both a `rect` used for determining the position of the ship and an image to represent it. In addition to a method for movement, a method for firing exists, which creates a projectile object. The projectile

class possesses the same fields as the ship, but lacks the method for firing.

A significant portion of the user's attention is drawn to the paths of moving projectiles. Our improvement upon this system is to do away with the projectile in favor of adding a weapon class which possesses a method to instantaneously draw a laser beam on the screen and register damage to the enemies hit by it. This instant damage allows the user to hit fast-moving targets by virtue of reaction time rather than by prediction projectile and ship movements. To introduce extra challenge for the user, both the ship and weapon class have color as a property. When registering damage, a weapon only damages a ship of matching color. A ship possesses a list of weapons and a method to switch the active weapon and fire it. This addition creates an imperative to match their ship's active weapon to the color of the enemy they are firing upon.

An open-source clone of Galaga which we chose to examine and improve upon is Pylaga. Pylaga is an implementation of Galaga written in python, and using pygame. It is, therefore, an easy system to compare to our own. Pylaga's enemies and players lack our concept of color. Pylaga's class structure is as follows:

It is clear from the above diagram that the author did not implement class inheritance, but used classes just for identification purposes and for storing some functionality. Much of the actual data was not encapsulated, but stored in global variables. Our version will be a marked improvement upon this. In particular, objects on the screen will inherit from the `RenderedBase` class, and moving objects will inherit the relevant code from `ActorBase`. This will lead to greater maintainability and core re-use.

The author of Pylaga implemented a player shooting using the following method:

```
def shoot(self, shotslist, locx, locy):
  self.boom = Bullet(shotslist)
  self.boom.set_pos(locx, locy)
  shotslist.add(self.boom)
```

This is a rather messy method, as it alters the state of its parameters, rather than updating a member of the class. Our proposed algorithm, below, will be a substantial improvement upon it.

For rendering images, the author of Pylaga chose to use global variables which are accessed and updated in each class's `update` method. The definitions and the update methods are:

```
# in globalvars.py
global playership
playership = []
# loads playership image
playership.append(
  load_file(DATADIR + 'pship.bmp')
  )
playership.append(
  load_file(DATADIR + 'pship1.bmp')
  )
playership.append(
  load_file(DATADIR + 'pship2.bmp')
  )
playership.append(
  load_file(DATADIR + 'pship3.bmp')
  )

# In player
def update(self):
  if self.state > 0:
    self.image = playership[
      self.state / explosion_speed
      ]
    self.state += 1
    if (self.state >= len(playership)
          * explosion_speed):
      self.state = 0
      self.image = playership[0]
```

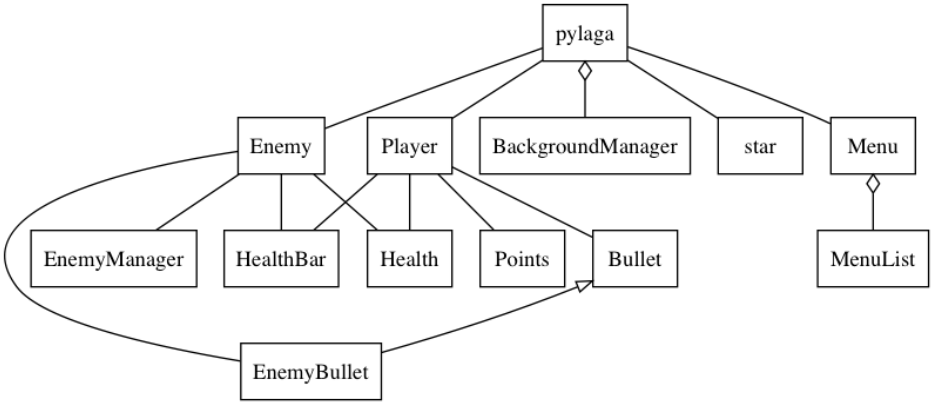

Figure 1: Pylaga's Class Diagram

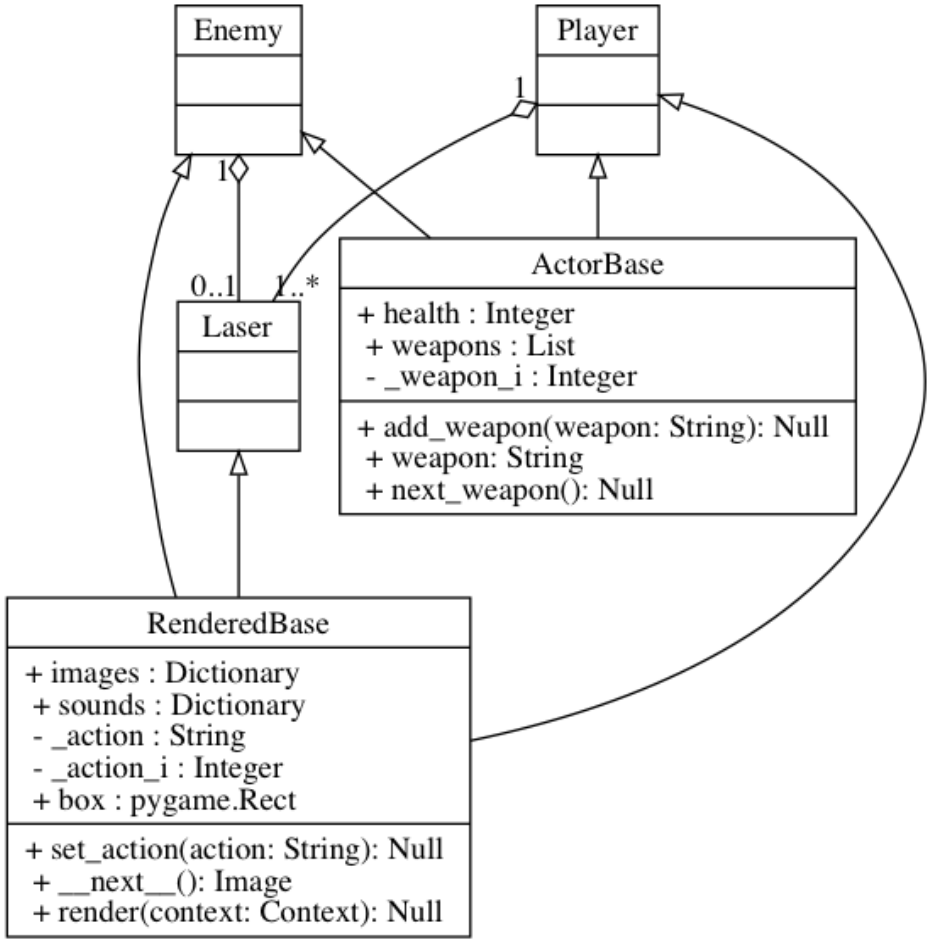

Figure 2: Our Class Diagram

The use of global variables here exhibits low cohesion among the components of the game. No other components but the player class are concerned with what image the player is represented with; yet all have access. Our system will leverage python's `__next__` "magic method" (also known as a "dunders" method) to increase component cohesion.

The Proposed System

Our proposed system is "Lazer Blast", an arcade-style shooter (like Galaga), which includes aspects of color matching and pattern matching similar to Simon.

Game-play

The user pilots a small ship against hoards of enemies. Each enemy has a given color. The player must use a similar-colored laser to destroy that enemy. For example, an enemy which is green much be shot with a green laser. A red laser will have no effect on a green enemy. Waves of enemies come in patterns: an astute fighter will be able to predict the next wave from the previous, and will have a significant advantage over the enemy.

Since the player cannot be expected to handle the copious masses of colorful enemies immediately, the difficulty of the game increases as the player progresses, starting with a single color and slowly adding other colors and increasing the speed.

System Architecture

Lazer Blast was designed using Object-Oriented Design, and implemented with an Object-Oriented Approach using Python and the game library, pygame. Python was choosen for the ability to rapidly prototype, as well as its widespread use among the open-source community. Object-Oriented Programming is well adapted to Python: Python, while being a multi-paradigm language, has a strong tendency towards Object-Oriented Programming. The framework, pygame, was chosen due to its relative simplicity and due to its relative impartiallity towards individual implementation choices.

In the Object-Oriented Design, all objects which are drawn to the screen are subclasses of the `RenderedBase` class. All objects which perform some action subclass the `ActorBase` class. For example, obstacles will be subclasses of `RenderedBase`, but all enemies will be subclasses of `ActorBase` and of `RenderedBase`. For the initial version of the game, there is only one obstacle, `Obstacle`, and only two types of actors: `Player` and `Enemy`.

In addition to the `RenderedBase` and `ActorBase` classes, there is also the `Menus` class. The few menu screens that there are will be subclasses of `Menus` and include the Main Menu, High Scores, and Options screens, which can each be built off of the `Menus` class by simply changing the graphics on that screen.

Algorithms

Gratuitous use of Python's magic method, `__next__` was made to implement all sequential algorithms that were necessary. In this way, each object which maintained a state through a sequence acts as a generator. This eases the complexity of logic, and makes maintaining state easier. (As the individual components exhibit looser coupling, and stronger cohesion.) For example, the images to be blitted to the screen for a given actor are defined as follows:

```
def __next__(self):
  if self._action_i == -1:
    raise Exception('Action must be set')
  self._action_i += 1
  return self.images[self._action][
    self._action_i-1
    ]
```

The GameSound class will implement the sound aspects of the game. It will provide the lasers with sound every time they are activated, will play a crashing sound when a specific colored laser hits the matching colored target and will provide a sound specific to the player losing or winning the game.

```
class GameSound(object):

  def _play(self, sound):
    def inner():
      pygame.mixer.init()
      pygame.mixer.music.load(sound)
      pygame.mixer.music.play()
    return inner

  fire = _play('laserFire.wav')
  hit = _play('directHit.wav')
  win = _play('playerWon.wav')
  lose = _play('playerLost.wav')
```

Our primary improvement over Galaga and Galaga-like clones is adding a color component. The actual logic for this component is simple. The LaserStrike class will display a laser object of a specific color when a certain key is selected. If the laser is of the same color as the enemy target and makes contact, then this class will call the GameSound class in order to implement the crashing sound that will return from a direct hit. This class will also call the Enemy class in order to deduct health upon a hit.

```
class LaserStrike(object):
  def _init_(self, color):
    self.color = color

  def enemy_color(self, enemy_color):
    if self.color == enemy_color:
      GameSound.hit()
```

As for algorithms that may be used in Menus, they are fairly straightforward. For the most part, the only input that a user can make on a menu is moving the cursor, and that is simply incrementing and decrementing a counter based on which direction the user inputs, and this counter is read in order to redraw the cursor on the screen so that the user can see which selection they have highlighted. The only other input a user can make is selection, which moves the user from the current menu to another screen, or closes the application. The current menu is assigned by a variable to know which graphic to render for the user.

Implementation and Testing

The base classes for *Lazer Blast* were completed using the Test-Driven Development paradigm. A total of 12 unit tests were written to implement the base classes, giving a solid foundation for the rest of the program. The unit tests in question used a stubbed subclass of RenderedBase to allow unit testing:

```
class Pantomime(RenderedBase):
  images = {
      'walk': ['fake_walk_{}'.format(x)
               for x in range(10)],
      'run': ['fake_run_{}'.format(x)
              for x in range(10)],
  }
  sounds = {
      'bump': 'fake_bump'
  }
```

The stubbed class and examples from unittests also helped develop a contract for creating subclasses at a later point, and so were successfull.

For instance, a unittest describing that actions are required provided an example for the later implementation of the actor class:

```
def test_no_action_exception(self):
  pant = self.Pantomime()
  with self.assertRaises(Exception):
    pant.set_action('skeddadle')
```

Difficulties in testing highly components were encoundered. For example, when testing the render method for the `RenderedBase` class, it would be expected to rely upon `pygame.draw.rect`. That is, when rendering to the screen, it would be expected to render as rectangle. To test this, it was necessary to using Python's standard `mock` library:

```
def test_render_called_with_box(
        self, mock_draw):
  surface = pygame.Surface(
      (1000, 1000)
  )
  rb = RenderedBase()
  rb.render(surface)
  self.assertEqual(
      mock_draw.call_count, 1
  )
  positional, _ = mock_draw.call_args
  self.assertEqual(
      positional[2],
      rb.box,
  )
```

The `GameSound` class is called when sounds are to be implemented during specific points within the game. This class works by submitting a string to the class through either a parameter or assigning the string within the class. For testing purposes, we have used the string loss which activates two sounds, one of a pilot saying mayday, mayday and another sound of the ship crashing. During the creation of this class, we ran into issues with successful testing as at first the sound did not play through. After some research and modification of the code, we quickly realized that the issue causing this was that we had forgotten to call `pygame.mixer.init()`, which initializes the Pygame sound and allows us to use the built in modules in order to play sound files being loaded in. After adding this call, we had once again attempted to run this class, using the pre-assigned string to activate a certain sound file. While the sound affects did come back this time and were correct, there was still an issue with the time gap between the first sound file of the pilot speaking and the second sound file of the ship crashing. This issue is one which we had expected to occur prior to testing and the reason which we chose the string, loss. We knew that this string had two sound files linked to it instead of one and that these sound files would both need to play and within a certain timeframe of one another. We wanted the first sound file to be cut off and the second sound file to start playing right before the first would end should it play all the way through. In order to handle this issue, we called `pygame.time.wait()` and had to adjust the number of milliseconds that we enter into the parameter so that our first sound file would had enough time to play through the piece which we wanted to use and so that our second sound file would not start playing until the right moment. The time frame we chose gives the player a sense that their pilot is calling for help as they are going down but crashes during the help call and therefore indicates a loss for the player. After some back and forth testing, entering a different timeframe within the `pygame.time.wait()` module, we were able to finally find the per-

fect timeframe so that we could accomplish the sound affects we desired for a loss within the game.

The `LazerStrike` function is one which activates the image of a particular colored laser striking as the correct keys are pressed. The way this function works is the player hits a certain key and based on the key they hit, the laser will display in either red, green, blue, or yellow depending on the key hit. These colors are ones which will be used for our enemy ships as well, which is why the key selection is important because if the player selects the wrong key then the colors will mismatch between the laser and the enemy ship and therefore the players attempted hit will miss. Another color which has been added to this function is white, this color is called anytime a key within an assigned event is clicked which will automatically be calculated as a miss for the player. When testing this function, there were several errors which we had run into with the first being that the laser line did not want to show up on the screen at all because we had not used the `pygame.KEYDOWN` module which is used to listen for when the player has a key pressed down. Once this was entered into our function prior to key events being assigned, then we realized that we had to adjust the position of the laser as it was going all the way from the top of our screen to the bottom. To fix this we simply played around with the start and end positions entered into the drawing module which gave us a better overall position of our lasers display. The positioning of our laser drawing is likely something else which will need adjusted again later as we get our final positions of the players ship and the enemies ships as we want our laser to look as if it is coming from our ship to the enemy ships. One way which we could do this is by assigning the players ships position to the `start_pos` variable and the targeted enemy ships position to the `end_pos` variable which we can pass to our laser drawing module.

During our testing of this function we had also realized that we were calling the wrong module when checking for which key was being held down as we were using the event.type module instead of the event.key module to assign keys being listened for. Once we had assigned the correct module to each key listening event statement, then our keys correctly called the color which we had linked them to. Our next problem was that the laser did not disappear after being called, it stayed on the screen and simply changed color. The ideal fix for this would be to completely remove our laser image, which were hoping to implement prior to our final release of the game. Unfortunately, we have yet to accomplish this fix but we were able to apply a fix where the laser would only show up as a visual to the player when they were holding down the assigned key. The way in which we were able to fix this was by implementing the `pygame.KEYUP` module which listens for when no keys are being used. At this point the laser color is assigned to match background color of our game. We can then make it so that this color has no effect, should this be our ultimate fix. Assigning the color of the laser to the same color of the background while no keys are being pressed, indicates to the player that the laser is not active. If there are multiple colors being used almost back to back, there would still be that flash inbetween keystrokes of a laser not being activated which will provide the player with the sense of the lasers being changed from one to the next instead of showing a constant active laser which only changes in color.