Milestone 4

Clay Asato, Jaskaran Buttar, William Chargin, Gage O'Neill, Mateus S Rodrigues

December 5, 2014

1 Introduction

Fancy yourself the pinnacle of evolution? Think again.

You are a microbe on Europa, a moon of Jupiter. Beneath Europa's surface is an ocean of ice and liquid water, teeming with microorganisms. Humans have sent a lander to search for signs of life on Europa, which you observe when a geyser propels you above the surface and gives you a glimpse of the lander's presence. You must ascend through the depths of the ocean, while at the same time growing large enough to be seen by humans.

Turn by moving the mouse, and move forward by pressing W. Fire on enemies with the D key, or activate a shield by pressing A. Destroy your enemies to absorb their DNA, which can make you either grow or shrink. You will have to grow to a certain size to complete a level. If you touch an enemy, you will have to restart the level.

Between levels, don't forget to spend any upgrade points you have at the shop!

Press J at the main menu for a special surprise.

2 Design

Our game's core mechanic relies heavily on conditioning the player. The player shoots at enemies and items are dropped. Green items make the player grow and release a green particle effect, and also increases the completion bar at the bottom of the screen, encouraging the player to repeat that action. Conversely, red items cause the player to shrink while releasing a red particle effect, discouraging the player from doing that action as they lose progress for the level.

Many aspects of our game contribute to the descriptive meaningful play. As the player collects the items to grow and complete the level, visual feedback appears instantly in the form of a particle effect, an animated level progress bar, and, of course, an enlarged player. When the player fires a shot, recoil pushes the player backward, making it clear that the player's action has had an effect; when the shot hits an enemy, the enemy is destroyed in an explosion cloud.

Some of the levels are specifically designed to force the player to strategize. This, in turn, generates meaningful play by forcing the player to make strategic choices in order to complete the level. For example, one level features a moving enemy spawner, and requires that the player destroy a high number of enemies to complete the level. By simply running around

the screen and shooting at enemies, the player will find it very difficult to complete the level. Instead, the player can observe the behavior of the spawner (notably, that it ceases to spawn once there are five existing enemies), and take advantage of the spawner's homing behavior to lure the spawner to a corner, then dash away and pick off enemies with the assurance that no more will be spawned in the meantime. Throughout our playtesting, we observed that this was one of the most rewarding levels for players to complete, because it was difficult yet achievable, and required more than mindless button-mashing to overcome. As such, we tried to incorporate similar design characteristics into other levels.

We designed certain aspects of our game to form feedback loops. For example, the core mechanic of the game requires that the player collect power-ups to grow to a certain size. However, as the player grows, it becomes more difficult to maneuver, fit through tight spaces, and avoid enemies. This is a negative feedback loop: as the player improves, the game becomes more difficult. We designed this feedback loop to stabilize the play, preventing players from speeding through the game.

The upgrade system in our game is an example of a positive feedback loop. After each level (except for the tutorials), the player gains an upgrade point, and an additional upgrade point for defeating a mini-boss. The player can choose to spend these upgrade points at the shop between levels. There are three categories in which the player can invest upgrade points (speed, shot range, and recoil), so the player has the freedom to choose an evolutionary path. This forms a positive feedback loop because, as players progress through the game, they develop their character, which makes completing levels easier. However, we also designed the levels to become increasingly difficult as the game goes on. This way, the increased difficulty and increased player capacity balance each other out, and the relative difficulty level stays approximately constant.

To give a sense of flow to the player we had to create the illusion of control for this game that would get the player to invest hours upon hours in the game.We did this by including a upgrade system for our player. By adding a upgrade system, it gives the player freedom of controling what he wants his player to become or turn out to be. For instance, if the player wanted to upgrade the recoil of the gun then his player would become really accurate, whereas if he increases his range he becomes a longshot sniper. In addition to upgrades, we added sound to give the player sense of control. We added sounds for killing enemies and collecting power-ups so it would help give the player a sense or feel that he actually did somet

Also, to make the game engaging for the player we made it pretty difficult. We believed if the game was to easy the player would get bored easily and see it as a little kid's game. So, to achieve this, we had to playtest our game a lot to find the perfect criteria for the game that would require you take many attempts, but at the same time wasn't impossible.

To create the sense of transformation of time and loss of self-consciousness we wanted to make our levels of the game replayable and wanting to keep the player playing till he completes the levels. So, we added the instant reset design to all our levels. Basically what this does is that when you die in the game, It instantly resets you to the start of the level. This allows the player to forget about the outside world and get lost in the game because it doesn't allow for any breaks where the player can reevaluate his decision to keep playing or give up. All that he/she can focus on his getting past the irratating levels. We borrowed this concept from the game, "World's Hardest Game." Because I noticed that when i stopped playing that game I had wasted a lot of hours in it and died about 500 times!

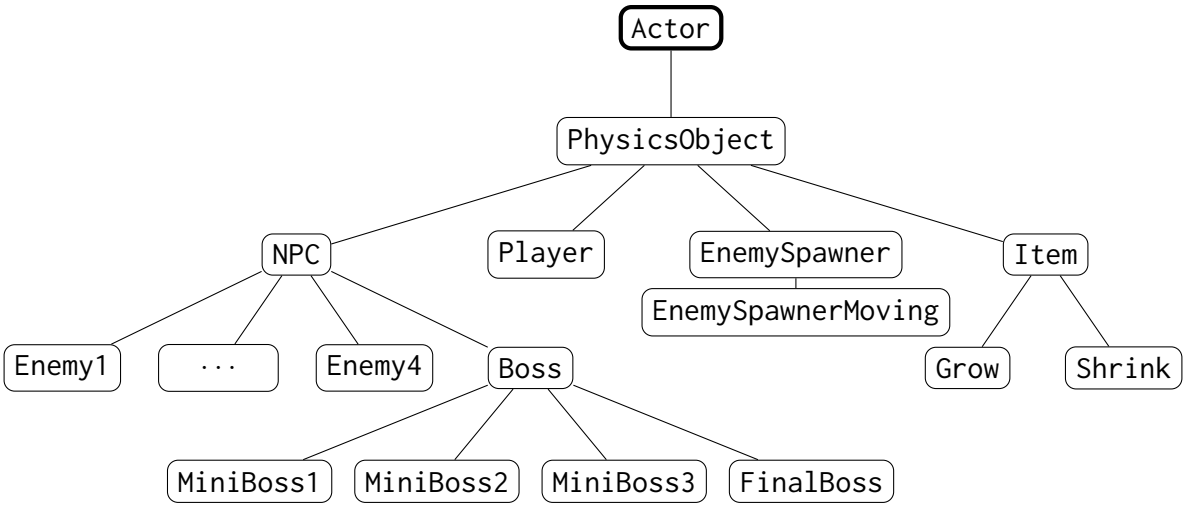


Figure 1: Diagram of a subset of the organizational structure for the physics system. Other subclasses of `PhysicsObject` include `Particle`, `Shot`, `Shield`, and more.

One of the improvements we made from our prototype to our final game to give a better sense of flow was create clear goals for the player to recognize. In our protoype our goal was a bit foggy because all we said was grow. We didn't really have a certain criteria or say how much to grow. So, to counteract this, we state the goal to each levels at the bottom of the screen for the player to see. For example, "Grow 4 times" this lets the player realize he needs to collect four powerups to comple the level.

One final aspect we added to increase replayability was a statistics screen. By pressing the S key at the map screen, the user can see statistics about total time played, death count, and shot–kill ratio. All of these provide objective metrics that incentivize the user to improve in subsequent game runs.

3 Implementation

3.1 Organization

Most of the objects in our game levels are subclasses of the `PhysicsObject` class, which we created to work as a physics engine. This class stores both position and velocity, as well as angular velocity, and supports a number of other physical effects (such as damping/friction and collisions). All values are stored as double-precision floating point numbers, unlike Greenfoot's built-in `Actor` class.

One major benefit of having a single root object for all physics objects is that it provides a unified interface that all classes can rely on. Many different aspects of the game, as written, work for any subclass of `PhysicsObject`: for example, any `PhysicsObject` can bounce off walls, or be reflected by shields (provided that it implements the `Reflectable` marker interface). This has the effect of letting many major features "just work"—for example, the particle effects emitted when the player picks up an item are true physics objects, and can interact with the environment. Another benefit is that it reduces the number of special cases

required. For example, shots can ricochet off of any number of walls before disappearing; no explicit relationships between the `Wall` and `Shot` classes are needed for this to be the case. Finally, the ease of use of the `PhysicsObject` class makes it trivial to create many different behaviors. Particle effects are simply random polar velocities with a fixed damping; falling objects simply have a positive y velocity; enemy homing behaviors can stack with normal motion because the physics system is unified; etc. This simplifies a great deal of game code that would otherwise be devoted to checking special conditions between pairs of classes, which in turn reduces bugs and speeds up development.

The other main global component is the `PlayerData` class. This class represents certain elements of the global game state that need to be passed around between levels, where the actual `Player` instances may differ. Specifically, the `PlayerData` class tracks which levels have been unlocked, how many upgrade points the player has earned, what upgrades the player has purchased, etc.

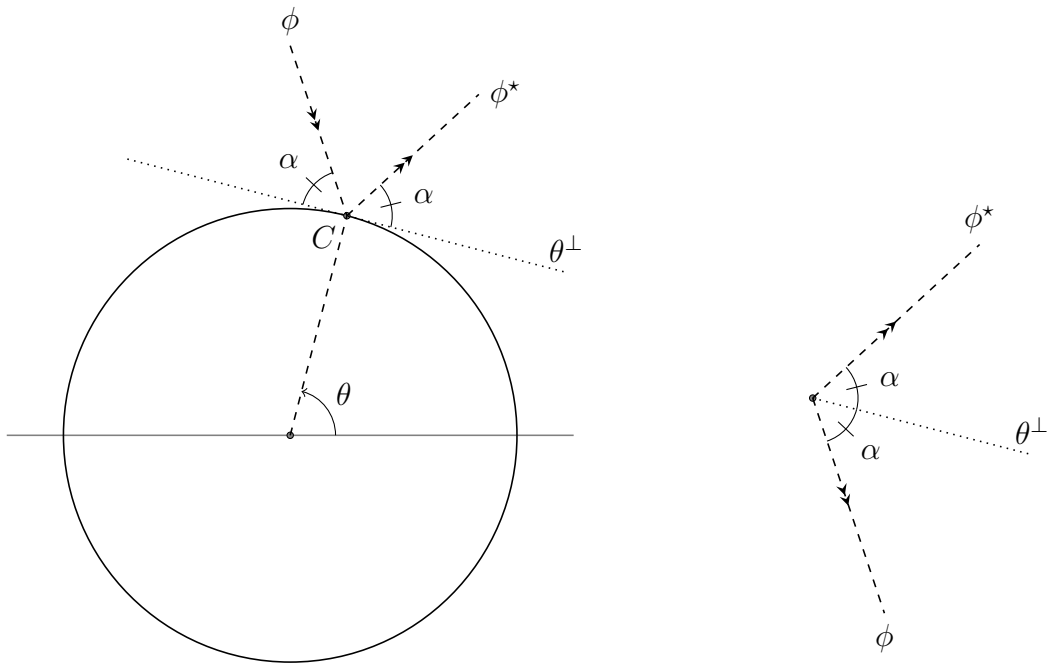
Beyond these classes, we have a number of sub-hierarchies under `PhysicsObject`. The `NPC` class is the superclass of all enemies and hostile obstacles; `Item`, of all powerups and upgrades that the player can obtain; `Shot`, for all types of projectiles; `EnemySpawner`, for all objects that can spawn enemies during gameplay; etc. We also employ marker interfaces to define characteristics of some objects: for example, any instance of a class that implements `Reflectable` can be reflected by a `Shield`. In general, we perform collision tests and other similar operations on the root of each sub-hierarchy to avoid repetition of code: for example, the player checks to see if it is in contact with an `NPC`, not an `Enemy1`. As such, we exploit polymorphism to simplify our game code.

3.2 Game Objects

One of the major in-game objects is the shield. Because the shield is a perfect circle, we were able to implement precise collision detection and reflection; the relevant mathematics are shown in fig. 2. Figure 2a shows the basis for our collision detection formula; for a projectile traveling along an initial trajectory with angle ϕ, and impacting the shield at an angular measure of θ from the horizontal, the angle of reflection should be $\phi^\star = 2\theta^\perp - \phi$, where $\theta^\perp = \theta \pm \pi$. The derivation is shown in fig. 2b. For collision detection, we filter the list returned by the built-in method `getObjects(Class<?>)` by their distance from the shield center (given by `Math.hypot(other.getX() - getX(), other.getY() - getY())`). This allows us to employ a precisely bounded hitbox instead of the standard axis-aligned bounding box.

Another important game object is the wall. Detecting collisions on a wall is more difficult, because objects can collide on either of two axes (the length or the short depth). To determine which axis is hit, we execute the following procedure. First, we construct a vector $\vec{r}$ from the center of the wall to the center of the colliding object.[1] Then, we construct a unit vector $\hat{u}$ along the length of the wall. By calculating the projection $d = |\vec{r} \cdot \hat{u}|$, and comparing it to

[1] In fact, even this is a slight simplification. We actually construct $\vec{r}$ as the vector from the center of the wall to the center of the colliding object *at the previous frame*. Without this adjustment, objects hitting the side of the wall would first clip through the wall, causing them to be erroneously detected as having collided with the front face of the wall.



(a) Path of a projectile initially traveling along trajectory ϕ, impacting shield at contact point C with tangent line $\theta^\perp$, and then bouncing off along trajectory $\phi^\star$.

(b) Selected elements from fig. 2a, displayed in standard position.

Figure 2: Explanation of shield bounce dynamics.

the known length of the wall ℓ, we can determine the axis based on

$$\text{axis in contact} = \begin{cases} \text{length axis} & d < \ell/2 \\ \text{depth axis} & d \geq \ell/2. \end{cases}$$

Then, we can simply reflect the incoming object about the appropriate normal.

A third important object is the enemy spawner. At a basic level, this object generates and launches new enemies. Each enemy spawner can be customized in the following ways:

Type of enemy spawned We have four primary enemy types: `Enemy1`, `Enemy2`, `Enemy3`, and `Enemy4`. These classes are all subclasses of the `NPC` class (see fig. 1). The `EnemySpawner` constructor takes an integer parameter **`int`** `enemyType`. Whenever an enemy is created, a series of **`if`** statements are executed to create the correct enemy. For example:

```
NPC e = null;
if (enemyType == 1)
{
    e = new Enemy1();
}
else if (enemyType == 2)
{
    e = new Enemy2();
}
// etc.
```

Launch parameters Both the angle and velocity of launch can be configured. The launch velocity is simply set by `setVelocity(double)`. The launch angle is configured with two methods: `setAngleStart(double)` and `setAngleRange(double)`. When an enemy is spawned, it will be launched at an angle sampled from the uniform distribution on $[\theta_{\text{start}}, \theta_{\text{start}} + \theta_{\text{range}})$. Special cases include when $\theta_{\text{range}} = 0$ (in which case there is no randomness), and when $\theta_{\text{range}} = 360°$ (in which case the enemy is fired at a completely random angle, and the value of θ_{start} is irrelevant).

Homing behavior In addition to the `EnemySpawner` class, we created a `EnemySpawnerMoving` class (which **`extends`** `EnemySpawner`). Instances of this class will additionally follow the player around the screen. The movement speed is configurable by the `setMaxSpeed(double)` method provided by ancestor class `PhysicsObject`; by default, the maximum speed is set to 0.1 px/frame.

Number of enemies to spawn Every `EnemySpawner` has an **`int`** `replacementCount` field, representing the number of enemies left to be spawned. The `Level` classes that create `EnemySpawners` can modify this field as desired. Typically, this is done by selecting some fixed number of desired enemies n^*, and counting the number of enemies n on-screen at the current frame. Then the replacement count is given by $\Delta = n^* - n$.

3.3 Major Issues

One major challenge in this game was the implementation of boundaries. There was a significant amount of math involved to determine how boundaries would interact with objects like shots and enemies. More specifically, determining the angle of reflection required a lot of whiteboard sketching and trigonometry. Additional problems arose when implementing walls, as we had to make sure that walls would not bounce off of walls when placing them close to one another. As it stands, walls are still kind of buggy and do not work perfectly.

Some issues in terms of gameplay were that shots had a habit of destroying DNA drops instantly if they dropped in the right direction. To fix this, a timer was added making the drops invincible for a certain number of frames.

4 Iterative Development

The main focus of game development is the end user. The use of iterative design processes greatly aids in bridging the gap between what the end user would enjoy and the end product. *Europa* went through several design stages each followed by playtesting. This was to improve the game and cater to the wide variety of players that would eventually purchase and play the game.

4.1 Prototype

Our prototype included a sufficient physics engine including recoil and momentum. This was the first element that was written. It would be useful in simulating the microbial and fluid motion of single celled organisms and add to the overall "cellular" paradigm in which our

design was formed. Originally, the player's only functions were moving and shooting. The player would face the cursor and move in that direction when the W key is pressed. Our first iteration featured very few artistic elements and only contained placeholders. However, in later iterations as became available, more art was added. The prototype included three tutorial levels each with the intent of conditioning the player and helping acclimate them to the unique controls.

4.2 Playtesting

We set up each player with a laptop with the game loaded, showed them the run button, and let them play through the entire current version of the game. In early stages of the game when cutscenes were not complete, to supplement we gave the following background information (delivered verbatim to each player):

> This game takes place on Europa, a moon of Jupiter. You are a microbe, and you are competing with other microbes to become the most advanced organism. Your goal is to grow big enough to eventually be discovered by a human scientific mission searching for life on Europa.

Other than this, no direction was given. Players were expected to figure out controls on their own. We wrote down the questions players had during their experience, but only answered questions if repeated. We also observed the behavior of playtesters, and noted reactions.

After each playtester completed the game, we asked the following questions:

- Please rate the following items from 1 to 4:
  - Do you see potential in the game?
  - Do you like the story concept?
  - How well were you able to adapt to the controls?
  - How difficult do you imagine the game will become?
  - Is this a good level of difficulty?
- What are the top two features you would like to see added to the game?
- What did you dislike most about the game?
- Other suggestions or comments?

These questions have the purpose of extracting from the playtesters basic information about the game basic, such as: controls, difficulty and most important next steps that should be done. The playtesters were mainly college students, with different level of knowledge from previous games.

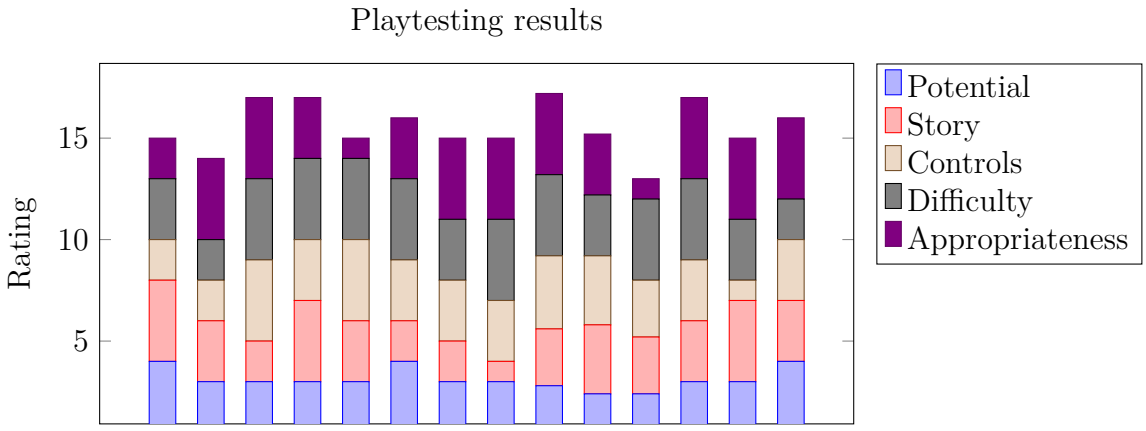


Figure 3: Playtester answers to numerical questions (each question out of 4).

4.3 Evaluation

After observing our testers during the playtesting stage, we compiled all the data and found several key trends in the gathered data.

First, in the initial stage of development, our game did not have art implemented. Lack of sprite and world art were one of the main concerns in the first iterations of *Europa*. Next, another trend in the data we collected was the dissatisfaction for the edge behavior. Previously we had simply ignored edges and allowed the player (and any subsequent enemies) to go far beyond the edges of the "world." Because of the fixed nature of worlds in our game, this opened up the potential for players to cheat or be unable to win. In addition, our playtesters were somewhat confused by our previously subtle approach to control instructions. In previous iterations of *Europa*, we had a simple image of a key flashing on the screen for several seconds before disappearing. Some players did not understand that it was a key simply based on the image, and others did not notice it at all. Later, after implementing walls and edge behavior, we reveived complaints about problems with wall bouncing. Throughout our own tests while developing the game, we came across the same problems.

4.4 Refinement

Based on these key trends in data and several more, we compiled a "to-do" list of refinements to implement, and carried out appropriate actions to remedy the problems we encountered during playtesting.

Implement Edge Behavior Implement edge behavior - after debating and wrestling with this topic, we came down to two different types of edge behavior

Edge-Wrapping When an actor goes off the screen, it will reappear at the other end of the with the same heading as in *Pac-Man* or *Asteroids*. We eventually threw this option out because of the future problems that would arise with enemy tracking. This edge behavoir would create a toroidal topology. Auto-targeting geometry would break.

Walls Edges would act as walls. There were several options for this: we could simply bounce back any objects disabling the player (and any other `Actor`s) from leaving the screen or slow down and eventually stop objects at the edge of the screen. We chose to bounce objects off the edge because of the possibility of degenerate strategies. If a player placed himself in a corner and held down the shot key, there would be no recoil and any tracking enemies would go right into the firing range. While this option slightly lowered the amount of usable space within the game world, it would work better with the elements we had already and were planning to implement later on.

Controls From our playtesting, it became apparent that our novel control scheme was difficult to pick up—and with good reason, because there was no instruction. Because players were having difficulty with the controls, we added in-game overlays describing each key as it became available. For example, in the first tutorial level, the player can only move, so a flashing "Press [W] to move" key appears, and an icon of a mouse with arrows follows the user's mouse. From this, players can quickly identify the control scheme. In the second tutorial level, the "Press [D] to fire" overlay clearly communicates this aspect of the controls, and similarly with the "Press [A] to shield" overlay in the third tutorial level. Once players had picked up the controls, they reported that they liked the control scheme and thought it was interesting.

Player Damage In the initial stages, our player was unaffected by enemies. Logically, we received several complaints about the fact that enemies did not kill or damage the player. We knew this had to be fixed, but did not know how to tackle this issue. We had two basic options: to set enemies to damage the player, or to set enemies to kill the player outright. After careful deliberation, we concluded that enemies should kill the player and the level should instantly reset. This was because of the fact that the player was shrunk or "damaged", by bad DNA. Bad DNA and an enemy bacterium should not be synonymous as far as damage goes. While this was later requested to be changed, we felt as though causing enemies to shrink the player would have made the game too easy. The level also resets immediately after player death, so the player has the opportunity to sharpen his/her skills with each death and eventually be able to complete the level.

Cutscenes A large complaint in *Europa* was the lack of cutscenes. While playtesters did not always mention this explicitly, it was implied by other verbal and non-verbal cues. Many players indicated that they did not understand the story and would have been unable to infer the entire story without cutscenes. We wanted to keep the game to a more minimalist nature, so we wrote some cutscenes that we thought would enhance the player's understanding of the game and create an emotional connection between the main character and the user. These cutscenes lacked audio (except the background music for the game) and dialogue. They only showed the player what we thought should have been shown and no more than that.

5 Future Work

To further enhance the graphics of our game a more convincing environment, we would like to implement some ripple effects in the future around moving objects to simulate movement through water. This would help make Europa's underwater environment more interactive. While this would be a very nice feature, it would be pretty tough to implement and may cause the game to lag. Thus we left the feature out.

Texts, menus, and fonts could all be made to look better on the main map screen, but as this was not necessary to the function or gameplay of our game we chose to leave updating these features for a later time.

These two features could combined to create an interactive menu in which the player image follows the mouse around on the menu screen to help the player become more familiar with the controls before they actually begin playing the game.

To help convey our story better we were thinking about adding cutscenes after every world of our game. We were thinking this would help add little breaks in the game to help explain the story to our player better and make it richer. We ended up scrapping this out of the game because of the time restrictions and also believed if we added to many cutscenes to the game it would break the overall flow for the player.

We also planned to add weapon upgrades, separate from the existing upgrade system. The design for these upgrades was that each miniboss would "drop" a weapon that the player could employ in subsequent levels. In addition to the standard weapon, we had planned to add three-shot spread, homing, and rapid-fire weapon types. This feature was deferred so that we could spend more time implementing more important features (like the minibosses themselves!), but would still be a nice feature to have in a future version of the game.

Along with grow and shrink power-ups we were thinking about adding a shield booster power up. What this would do basically was temporarly boost your use of the shield and allow you to use it for longer durations of time. We ended up scrapping out this power up because we belived that it would make the player over powered and make the game to easy disrupting the flow of the game. Also, we thought that too many power-ups on the screen would just distract the player from the main goal of the game which is to grow and kill.

6 Conclusion

All our work throughout the quarter on *Europa* has led to this, our final product. While it still has some kinks to be worked out, we believe it properly represents the effort of our team. Our game may not end up being the next big multi-platform blockbuster, but using the iterative design process, we were able to create a game that has vastly improved over the initial design in the beginning of October. Thanks to the patience and insight of real people who playtested our game, we gained insightful feedback that led directly to improving the game to cater to the needs and preferences of real people, exactly the types of people that would end up playing our game when we put it on sale. Throughout our experience building *Europa*, we gained valuable knowledge, skills, and confidence in programming.