SOLOBOT

GROUP
FREE BEER

Cory Gross
William Wallace
Jack Satriano
Carter Michaels
Joshua Kane
Dillon Daugherty

P1

General Game Description and why it is exciting

- **Description** : 2D side scrolling where the player must navigate from the start through the level to the final objective.

- This game will be exciting because we will feature unique gameplay elements such as reflective armor to reflect attacks back at enemies. It will be entertaining in gameplay as well as thought provoking as the player has to approach different situations with different solutions. We will also feature very tight controls in order to eliminate any player frustration. We will put an emphasis on skill and timing always driving the player to perform slightly better and constantly challenge themself. Also, our game will feature robots and robots are cool.

A. Players - Single player

B. Game Play Objectives - Escape from the robot factory that the player is trapped in (Start to Finish)

C. Procedures or Rules - The player has a certain amount of health and must avoid being hit by enemies in order to sustain positive health and stay alive. The player can eliminate enemies using his weapon, allowing enemies to hit and eliminate each other, or by reflecting enemy attacks using a shield.

D. Resource Conflicts - The player has to pick up health and ammo and faces time constraints.

E. Boundaries or Formal Elements - Platforms which the player must navigate through as well as the factory in which the player is trapped in. The player must overcome the robots and escape the factory.

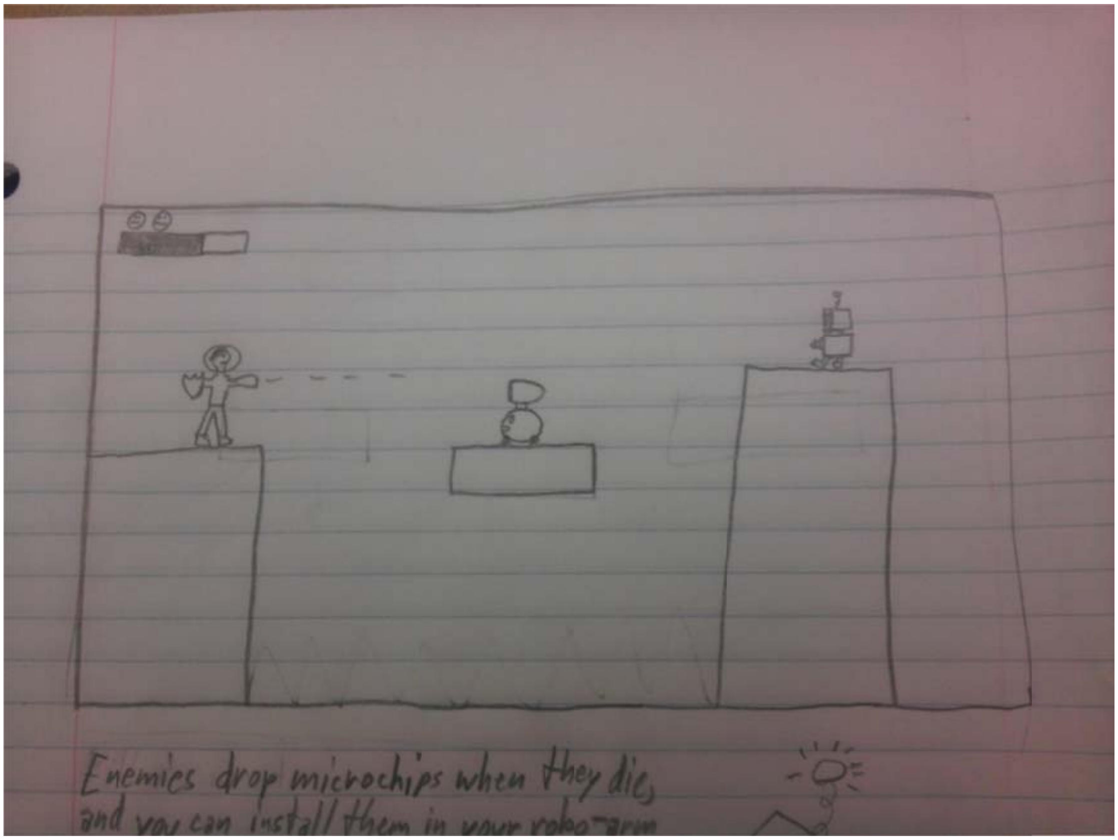
F. Outcome - The player escapes the robot factory or becomes trapped within and subject to robot experienments

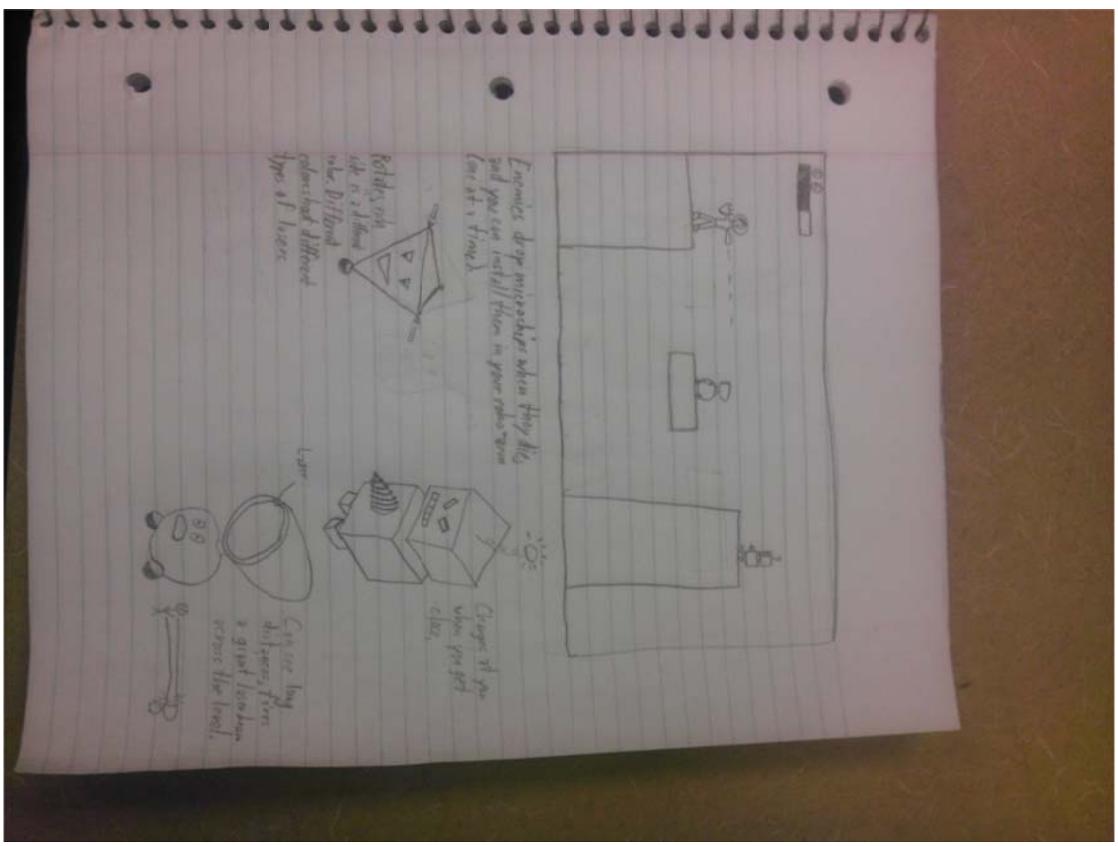
4. Development tools

- Platform : Unity
- Audio : Audacity
- Graphics : 3Ds Max

5. Development Levels

- Alpha : March 20th
- Beta : April 10th
- Final : April 20th


Enemies drop microchips when they die,
and you can install them in your robo-arm



P2

Design Decisions

Gameplay

The gameplay of Solobot is very similar to MegaMan and utilizes the jump and shoot base gameplay mixed with platforming. The player jumps over platforms and attacks enemies to gain "power-ups". These power-ups will change the functionality of the primary gun and change its power in different ways. The power ups will have pros and cons and will challenge the player to choose which power-ups will be most useful in any given situation. The vanilla laser will still be a very valuable asset that the player must acknowledge. Some power-ups will be more powerful against different enemies and weaker against others. These power-ups may be themed based on basic elements (Earth, Wind, Fire).

In terms of level design, the levels will be linear and last around 5 minutes long. The player must travel from left to right to reach the end of the level. There were various enemies that can be destroyed with the player's weapons. These enemies will drop power-ups as described above. There will platforms and pits that the player must traverse. There will also be spikes that will kill the player instantly. We do not expect a first time player to be able to finish a level on their first time. The levels will provide a level of difficulty that walks the line of frustration and challenging. When the player reaches the end of the level, he will be faced with a more powerful enemy (a boss). The player's health is partially recharged, and the battle begins. There will be a checkpoint before the powerful enemy to give them another chance at victory.

Controls

The controls for Solobot will have two different types of primary input. The default will be designed around the keyboard and utilize WASD and Arrow Keys (Up, Down, Left, Right) for movement. The WASD format is often used for games that utilize a mouse, but also leaves room for other keys on the other side of the keyboard such as Enter. We will use the space bar as a jumping key because it ergonomically lines up with WASD format. Right Shift will be used for shooting. This placement of keys allows jumping and shooting to occur simultaneously with two hands. The start button will be the Enter key so that the right hand can freely pause the game at any given time.

For the Xbox controller, we will map the controls much differently. The movement will be controlled by either the d-pad or the analog stick, this should be easy to implement because Unity has built in interfaces for controllers. The A button will be used to jump because it is the most common jump button, and corresponds with other games of this type for familiarity. The shoot button will be B so that the user does not have to move their fingers far. This keeps our "jump ‘n shoot" gameplay device in tact with the controller. The Xbox controller functionality will provide gamers with more precise control that is ergonomically fit to their hands. We will utilize Unity's built in libraries for controls to implement these ideas.

Power-Ups

- Static Field
    - This power will allow the player to project an electromagnetic field surrounding them that can hurt or stun enemies. Movement will be limited during use.
- Energy Deflector

    - Allows the player to use an energy shield that projects itself around the player and absorbs or deflects energy projectiles. Movement will be limited during use. If timed correctly, a deflection will occur.
- Overload
    - Allows the player to select and focus on a close enemy and overload their power supply, causing a small explosion.
- Power Jump
    - Allows the player to jump much higher than normal. Requires charging.

Enemy Types

- Spiked Spinner
    - Enemy that spins like a cap covered with spikes and tries to cut the player.
    - Stats:
        - Armor: Medium
        - Attack: Medium
        - Speed: Medium
- Stationary Turret
    - Enemy that spins like a cap covered with spikes and tries to cut the player.
    - Stats:
        - Armor: Low
        - Attack: High
        - Speed: N/A

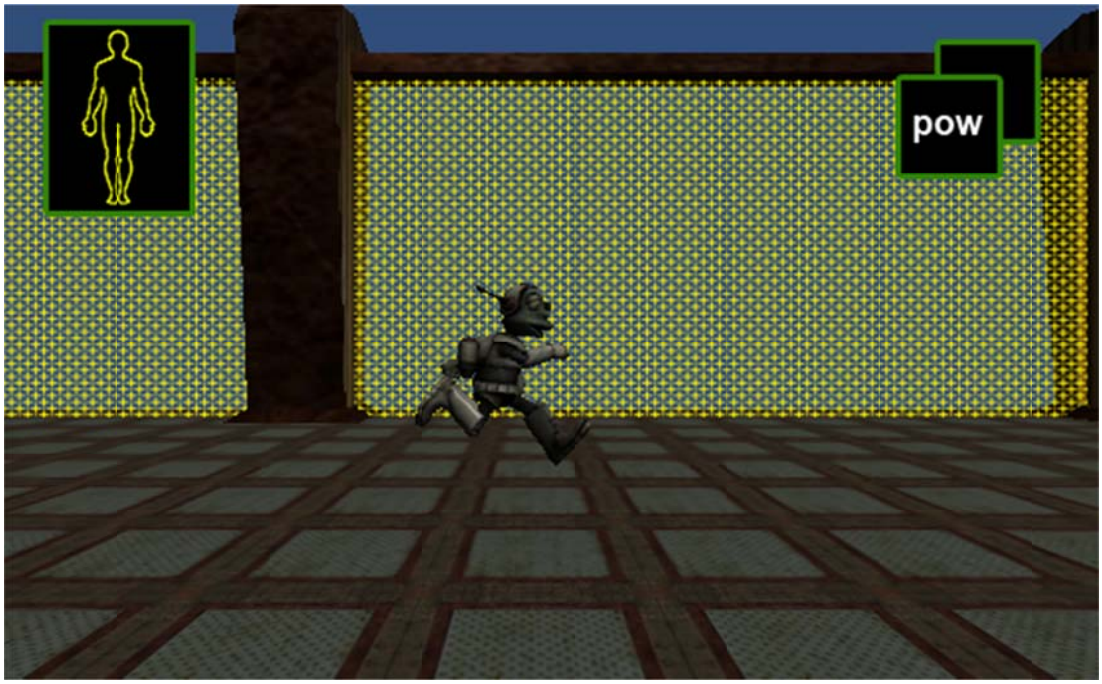
- Hover-bots
    - Floating support enemy that hovers near its allies and provides them with energy shields. Cannot attack on their own.
    - Stats:
        - Armor: High
        - Attack: None
        - Speed: High

Interim Report

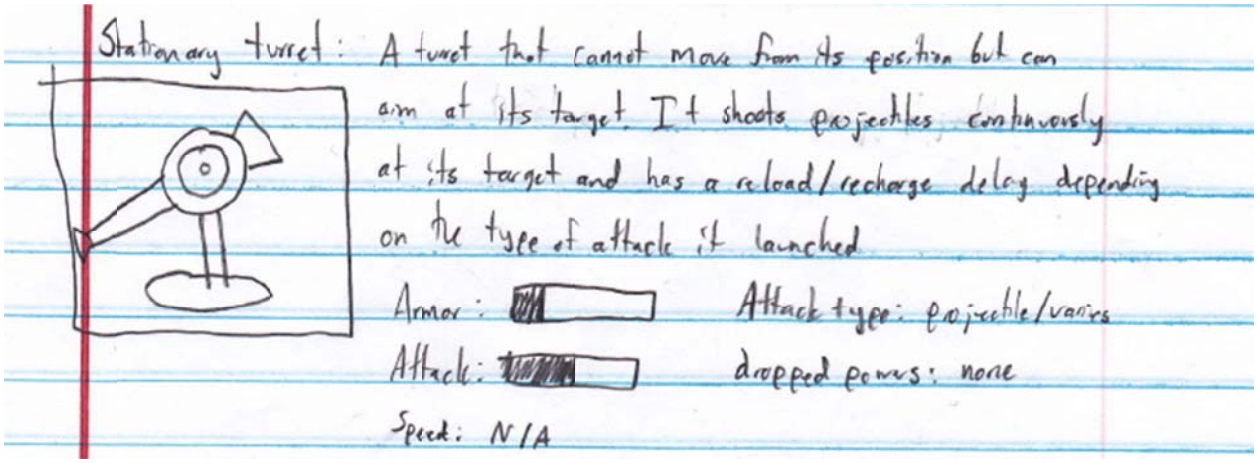
The current state of our game has reached our goals for layer 1. We have a working prototype game with a player character that can move around a simple level and perform basic actions like jumping and attacking. The player currently interacts properly with the environment and collides with geometry. The controls the user can utilize to interact with the game are easily changeable and will be merged with our current design for controls very soon. The Unity engine has provided a lot of help for us in several aspects of the game. The built in PhysX engine that Unity includes has been working very well for us and has made the character interactions with the world a lot easier than we had originally planned. The art assets have been more difficult than we originally thought, because getting everything to look fluid has been very difficult. Our current prototype is still using built in assets to keep everything working.

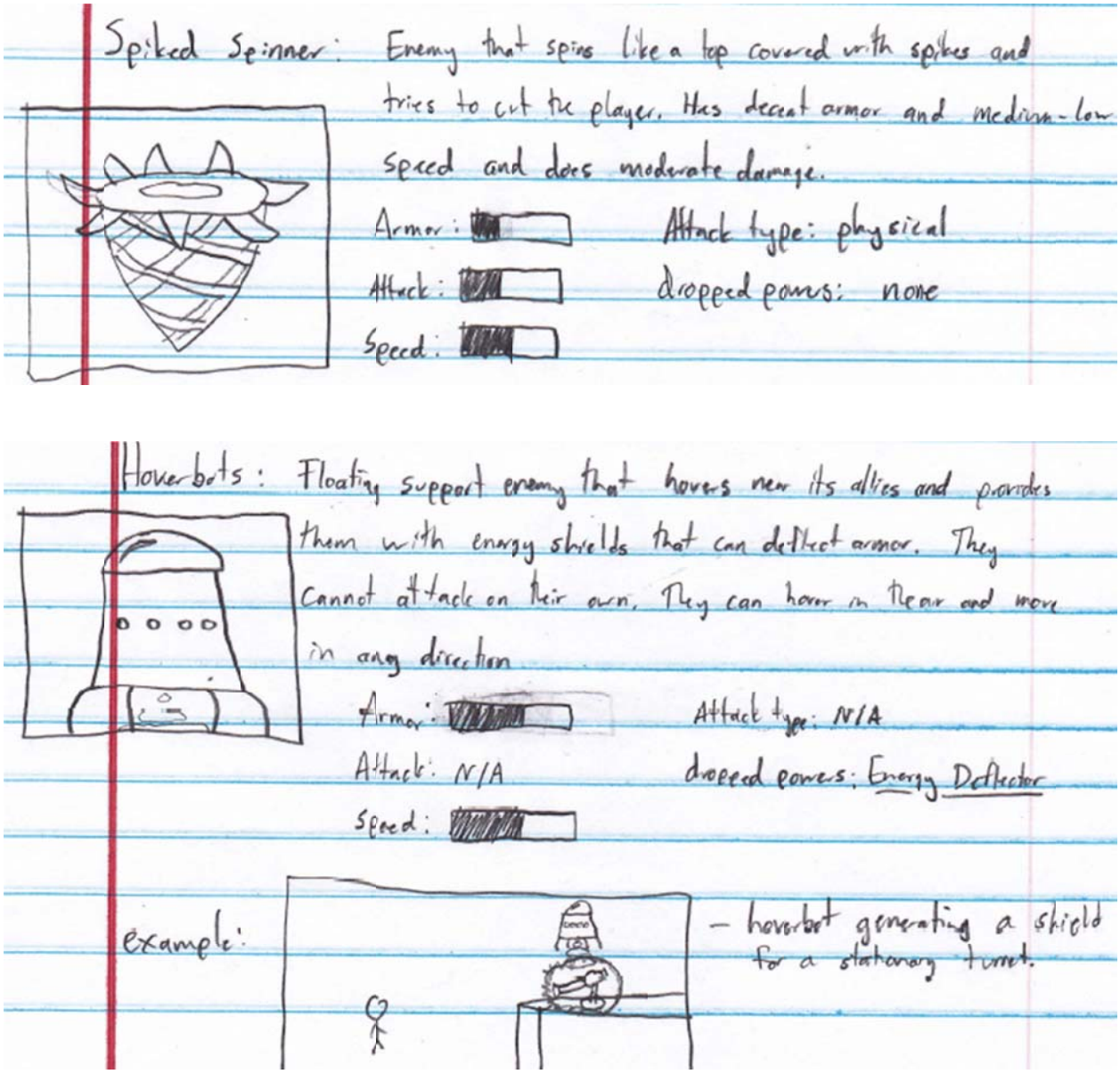
Design Sketches

Prototype Screen



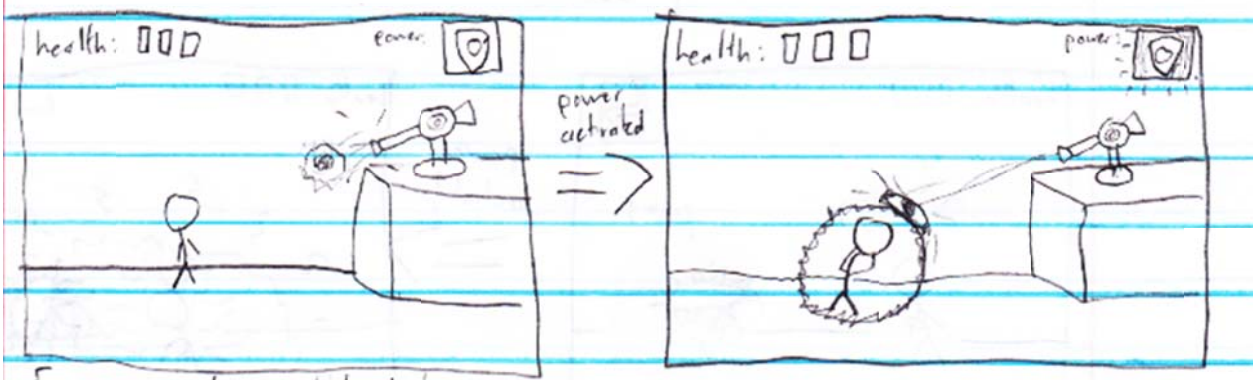
Enemies




Spiked Spinner: Enemy that spins like a top covered with spikes and tries to cut the player. Has decent armor and medium-low speed and does moderate damage.
Armor:
Attack:
Speed:
Attack type: physical
dropped powers: none
Hoverbots: Floating support enemy that hovers near its allies and provides them with energy shields that can deflect armor. They cannot attack on their own. They can hover in place and move in any direction
Armor:
Attack: N/A
Speed:
Attack type: N/A
dropped powers: Energy Deflector
example:
- hoverbot generating a shield for a stationary turret.

Powerups

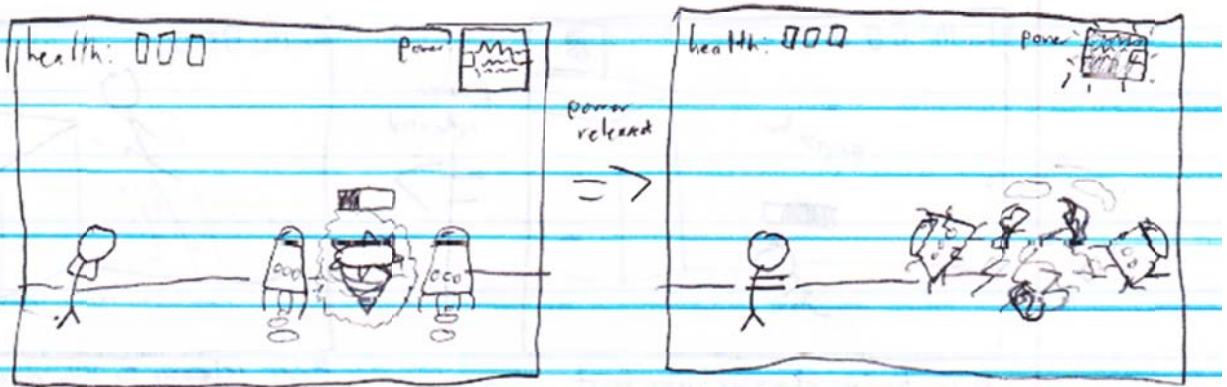
Energy Deflector: An energy shield that projects itself around the player and absorbs/deflects energy projectiles shot at it. This limits the player's movement and if timed perfectly, can reflect projectiles back to the enemy.



Enemy projectile is shot at the player

Projectile is stopped and can no longer hurt the player.

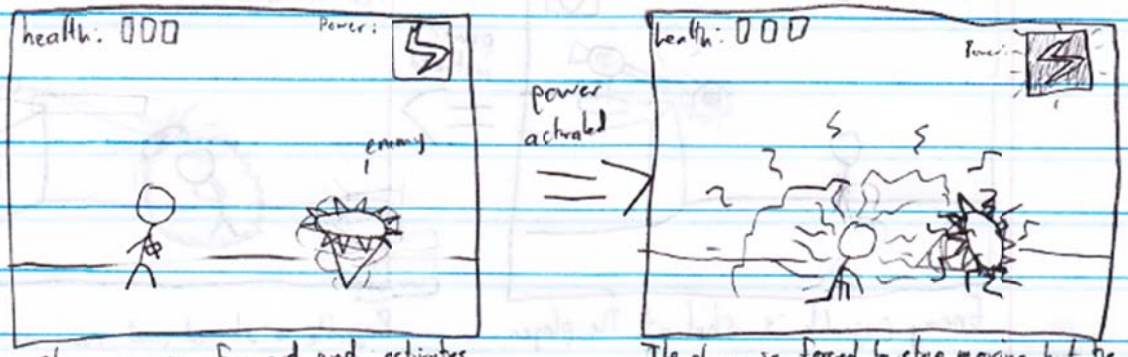
Overload: Power allows the player to select and focus on a close enemy and overload their power supply, causing a small explosion



Player selects middle enemy, begins charging the overload power

Selected enemy explodes, causing an area of effect explosion affecting all nearby enemies.

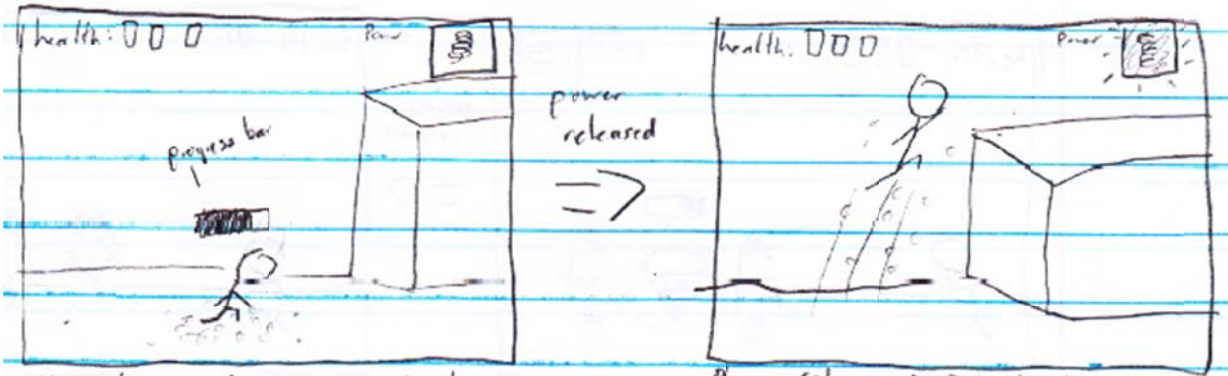
Static Field: -This power will allow the player to project an electromagnetic field surrounding them that can hurt or stun enemies. Movement should be limited during use.



player moves forward and activates power

The player is forced to stop moving but the enemy is either disabled temporily or destroyed

Power Jump: - When taken from an enemy, this power allows the player to jump much highe than normal. Requires charging.



Player begins charging jump boost power

Player releases the fully charged jump and is able to reach the platform

P3

Solobot - P3 Minimum Target

Progress in Unity

We have learned a great deal about the Unity Editor during our project. We started our development process by studying the Third Person Tutorial project that is downloadable from the Unity Asset Store for free. This project contains a sample player character model that we have used as a stand in until we are able to develop a more suitable replacement. Our lack of an artist has been somewhat of a hinderance. We started by identifying major scripts that controlled the behavior of the Third Person Demo.

Basic Scripting (Scripts of Interest from Third Person Demo Project)--

- **ThirdPersonController**
  - This script is attached to your third person player and controls the player object based on user input.
  - This is where the majority of your players behavior is handled. Since the tutorial was third person and we were interested in creating a side scroller, we used this as a guide, but we stripped out a lot of the behavior and locked the player to the XY plane.
  - This script controls the player object, but does not interact with the Camera at all. We called our new script SideScrollController based on this script.
- **ThirdPersonPlayerAnimation**
  - This script controls the player object's current animation based on data from the ThirdPersonController.
  - We created a `SideScrollPlayerAnimation` script that calls the animations provided with the sample character that are provided with the tutorial project.
- **Camera Scripts**
  - There were several camera scripts included in sample project, showing how to do a orbit camera based on mouse input, a smooth follow camera, and a spring follow camera.
  - All of these were suitable for a Third Person game but we needed something different for a side-scroller. We were able to write a simple script using the scripting language.
  - We use public variables accessible within the editor in order to change the camera's distance from the player, as well as the x and y offset. For now our camera's `Update()` method simply follows the x and y position of the players `tranform` component.

Basic Demo

At this point we were able to get basic side scroller functionality. Here is a **video screen capture** showing a demo featuring basic side scroller functionality after we completed the above programming based on the Third Person platformer tutorial included with Unity.

Additional Scripts--

- **ThirdPersonCharacterAttack**
  - This script is called `ThirdPersonCharacterAttack`, but it does not have anything to do with the fact that the game is ThirdPerson, we were able to use this as a guide to creating our our `SideScrollCharacterAttack`script.
  - This script is set as a component of the player object and works by monitoring input and the Character Controller component within its `Update()` method calls.

    - It also finds objects with the tag 'Enemy' and sends attack messages to them so that damage can be dealt and the appropriate actions can be taken. Simple enough.
    - This currently only handles a Melee attack, so we are going to have to develop either another script to handle our shooting functionality our add on to our script we created here.
- **ThirdPersonStatus**
    - This script once again has nothing to do really with the fact that this sample project is a third person example.
    - This script simply keeps up with the players state, we used it as a guide to create our own script that would keep up with our player's current state in our side scroller game.

Project Team Roles

We have a few various roles, but we have all generally contributed to the project.

- **Cory Gross** - Documentation, Website, Source Control, Programmer
- **Dillon Daugherty** - Design, Sketches, Programmer
- **Jack Satriano** - Story, Programmer
- **Carter Micahels** - Story, Characters, Programmer
- **Joshua Kane** - Documentation, Story, Programmer
- **William Wallace** - Sound, Programmer

As we are lacking anyone with formal skills in 3D modelings or as an artist, most of our progress has been technical, in scripting and functionality, rather than in original content.

Adding Enemies

The sample project we were studying also came with one sample enemy and some scripts that controlled it. It also controlled the animations. From this script we were able to create an `EnemyController` script which is the basis for the script which will control the AI of our enemies.

Here is a **video screen capture** showing a demo of how the script interacting with our player script. They work by sending messages back and forth, dealing damage to one another. You can see the hit spheres for each of them in yellow in the video. The enemy there had 3 hit points, we had a 6 hits points originally. Because these variables are able to be changed dynamically in the editor I was able to give myself more hit points while debugging the game.

P4

Update of Game Progress

Since the last time we showed our game we have implemented many features and tweaked others. We have added a nice a functional GUI system which shows our current power-up, health, lives, allows the player to pause the game, exit back to the main menu. The scripting in the game is capable of detecting the platform the game is running on. The GUI is adapted depending on whether the game is running as a stand-alone executable or in the in-browser web player. If the current platform is the web player then the normal exit button is replaced by a fullscreen button. Exit is not possible in the web player and an option for fullscreen makes more sense instead.

We created a whole new power-up which is the force field power-up. This is powered by a script called `ForceFieldController.js` which is attached to our Player object. The force field power-up uses the Shuriken Particle System editor which is new to Unity 4. The force field prefab object is always attached to the player and is only enabled when needed. The force field prefab is made up of a Particle System component, a sphere collider, and the `ForceFieldController.js` script.

The jetpack power-up was also changed, the height of the boost was lowered as well as an option to enable and disable the jetpack boosters.

We added the original music to the game constantly running in the background while playing. The music was developed by William on our own team. We created a main menu scene which is the first scene to be loaded in our game. We used free music found in the Unity Asset Store for our main menu background music. We created a star wars themed credits sequence which is accessible from the main menu.

Because our game was put under source control using Git and published to GitHub since the very beginning we can provide a link:

https://github.com/FreeBeerGames/Solobot/commits/master

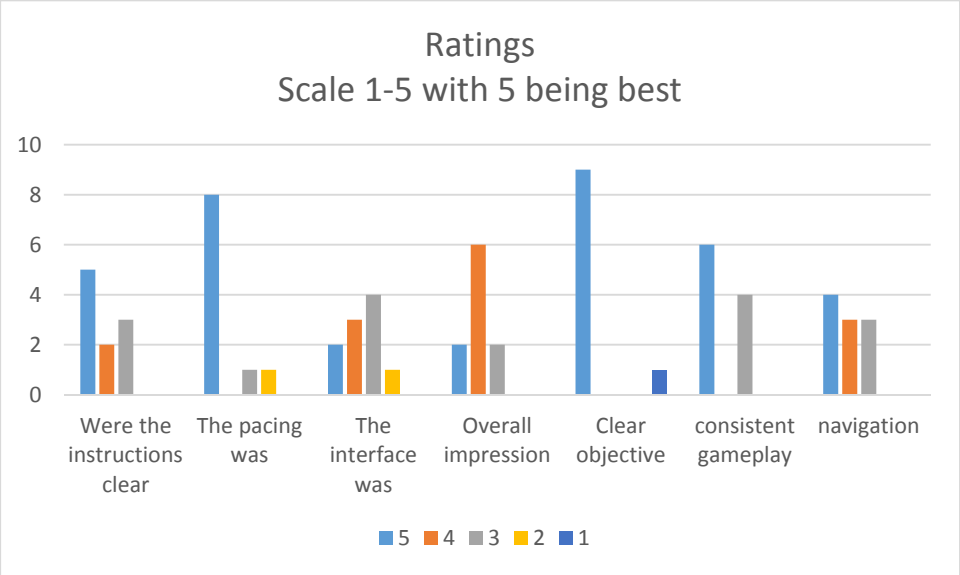
which details every commit to the project that was made starting at the beginning of the semester all the way up until the night before the final turn in.

Our entire game project is available from the following link:

https://github.com/FreeBeerGames/Solobot

Our game's website is also hosted on Github and is available as the gh-pages branch in our open source repo: https://freebeergames.github.io/Solobot

Play Testing Results



From these results we were able to determine that our instructions are mostly clear. The pacing of the gameplay was ideal. The interface could use some work but is mostly acceptable. The overall impressions weren't great. Our objective is very clear. The gameplay is mostly consistent and our navigation is mostly good. Overall we learned that our game was in good shape and the majority of users liked what they saw. The comments proved more useful than these ranking results and offered clearer suggestions on what the users would like to see.

Important Comments

- Game looked awesome compared to other people's games
- I really liked the character models and the overall interactions of the characters. Such as being able to shoot and fly around.
- Need ability to toggle jetpack on/off so I can jump without it.
- Maybe provide directions of what is supposed to be done. I didn't know the goal was to get out, I just felt it was to destroy everything.
- It would be interesting with more enemy types
- There were times I wanted to do short jumps with the jetpack off but the jetpack always activates when I jump
- I really enjoyed the game. The only issue I had was that the jetpack had a standard jump height. So even if I only wanted to jump little distance, it took me high into the air causing me to miss a lot of my landing points.

From the comments we were able to identify 2 main issues people would like to see worked on. One issue is that our jetpack was not as controllable as the users would like. We also learned that the users cannot see as much as they would like of the game world when they are flying using the jetpack making it hard to know where they are headed or where they are landing.

Lessons Learned

What we're most proud of

- **Particle Systems** - Working with Unity's particle systems to develop the jetpack effect, projectile energy balls and explosions for both player and enemy turrets. Shield power-ups.
- **Original Music** - The game's background music was composed by William and we felt it fit the design of the intended game very well.
- **Resourcefulness** - Lack of much in the way of visual artistic talent on the team and more technical members, we had to figure out a way to reuse what we could find for free in the way of assets online and focused our development on building a demo game with as much functionality as we could.

Changes to the original design

For the most part we did not deviate far from our original designs. We had less enemies than we originally wanted to have due to our lack of knowledge on creating models.

Things learned from play test and changes made

The jetpack power-up originally could not be disabled and was undesirable in certain situations in-game. We decided to have a power-up input which would allow the player to manipulate their power-up in some way. Pressing the "Power Up" button when the player has the jetpack either enables or disables the jetpack. Pressing the "Power up" button when the player had the force field it would activate it temporarily after which their would be a cool-down period.

What we would do if we had more time

Probably what we would do if we had more time to work on this demo is add some more enemy types, give the copper bot a more sophisticated second attack, firing and collision sounds for the turret, replace the Lerpz character model we are using for the player with something more in line with the plot of the proposed game.

What we would do differently

We would put more attempt into creating an original player model for our demo, given the time constraints we didn't feel like we could produce something suitable for our needs in time so we made due with one of the player models included in the free assets in the Unity Asset store. We learned a lot about how to create models and manipulate animations in 3Ds Max which will be an asset in the future.

Additional Documentation

We are including some additional documentation that we created for team’s tooling:

Creating Normal Maps with NVIDIA Texture Tools

Guide by Cory Gross

Introduction

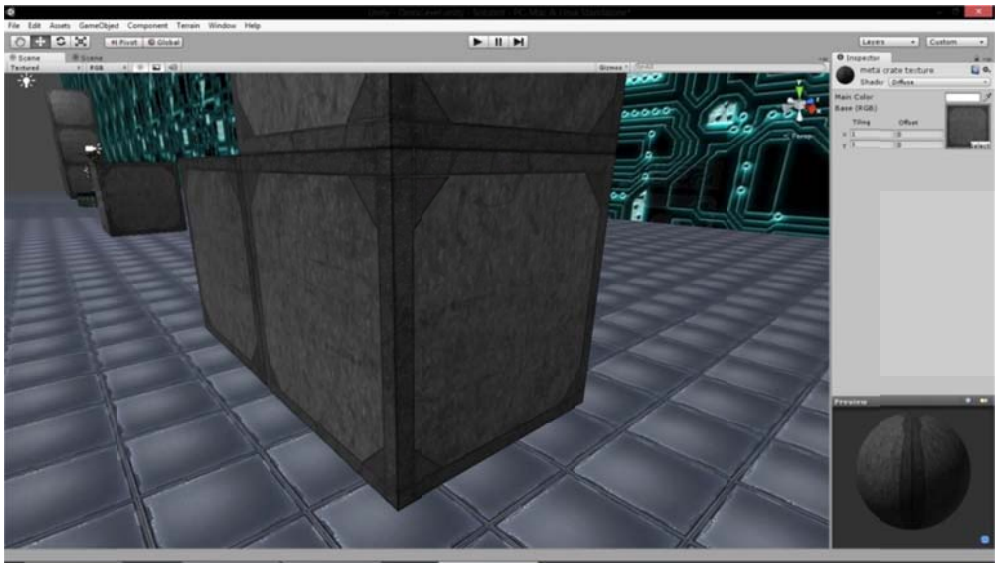
Texture is a large part of a human's visual experience. There are not many materials in the real world that are completely flat and devoid of texture. In creating games, many objects such as a wooden crate or a carpeted floor will be modeled using the minimal amount of polygons in the geometry such as using simple box for the crate or a flat plane for the carpet. Adding an RGB image to texture these objects helps to provide a bit of visual information about the material. However, because a simple cube is being used for the geometry it will still appear flat and when viewed at certain angles, lighting in a 3D space will give away the fact that there is only a picture of a textured material pasted onto a completely flat surface. This hinders the realism that can be achieved in a game.

One solution to this problem could be to make our geometry more closely conform to the actual object we're attempting to model. Instead of using a simple cube to model a crate, use many sizes and shapes of rectangles to represent the planks that make up the crate individually. Some of the planks will stick out farther than others. This is an ideal solution as we are giving our objects actual texture through its geometry. However, the more complex the geometry in our game, the less time you will have per frame in order to do other processing such as physics handling and AI controllers. UV mapping a texture onto the object also becomes non-trivial when the geometry is complex.

So Why Should I Care?

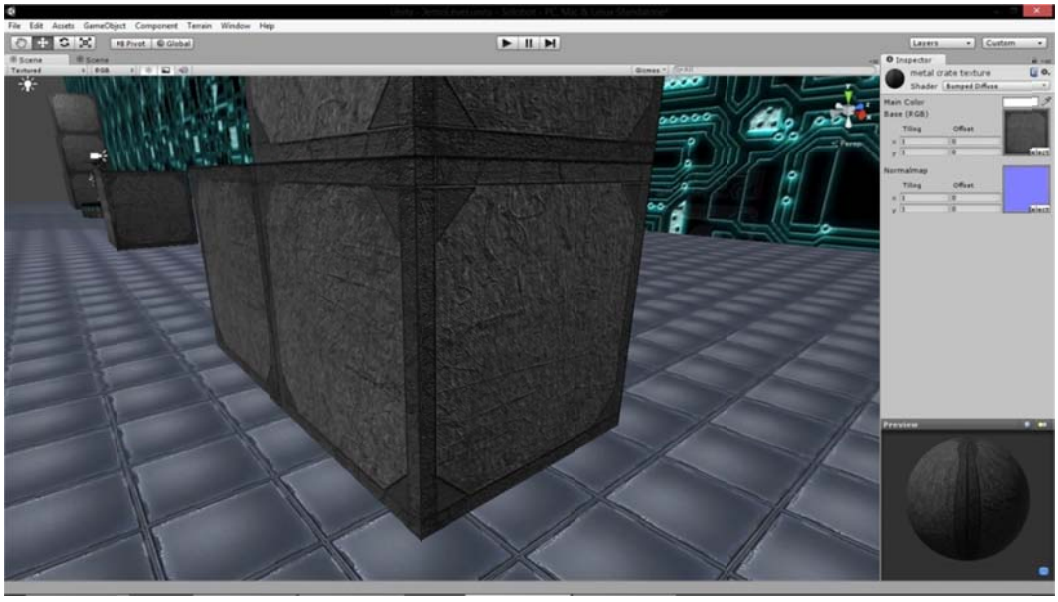
Instead of using incredibly complex geometry for modeling the surface of a textured object such as a carpeted rug, usually a technique called **normal mapping** is applied in order to mitigate the above problems. For my sidescroller demo Solobot created in Unity we found a scratched up metallic looking texture for our platforms and crates. We thought it looked good so we used it for our main platform texture. Here is what it looked like before applying a normal map:

Before Applying Normal Mapping To Crate



And this is what it looks like after using NVIDIA's Texture Tools in order to generate a normal map and applying it in Unity via the material's shader. In both screenshots only a simple cube is used for the geometry of the crate.

After Applying Normal Mapping To Crate



So What Is It?

A **normal map** is nothing more than a regular RGB image. In order to understand how it is used to create more detailed visuals in games you have to know how Lambertian (diffuse) lighting is calculated across a surface.

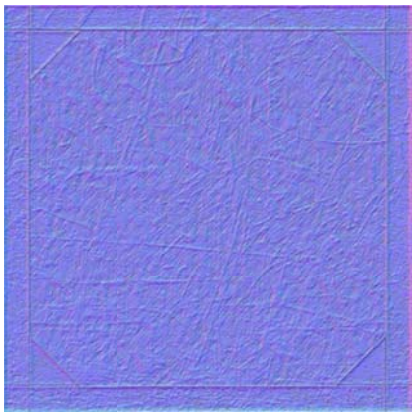
In order to calculate the **diffuse lighting** for a particular point on a surface given the position of a light source you take the dot product of the vector from the point to the light source and the normal vector from the surface at that point. In other words, if we have a light source at the point represented by vector **L**, and we have a point on a surface at **P** with a normal vector from the surface at that point of **N** then the intensity of the light source at that point is calculated as:

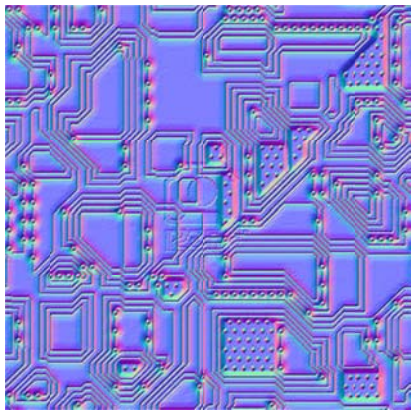
$$i = (**L** - \mathbf{P}) \cdot \mathbf{N}$$

This intensity `i` will always be a value between 0 and 1, with a maximum value representing 100% intensity and a value of 0 representing no lighting from that source. The intensity will be 1 if the normal vector and the vector from the point and two the light source are the same direction. If the two vectors are orthogonal (perpendicular) to one another then the value will be 0.

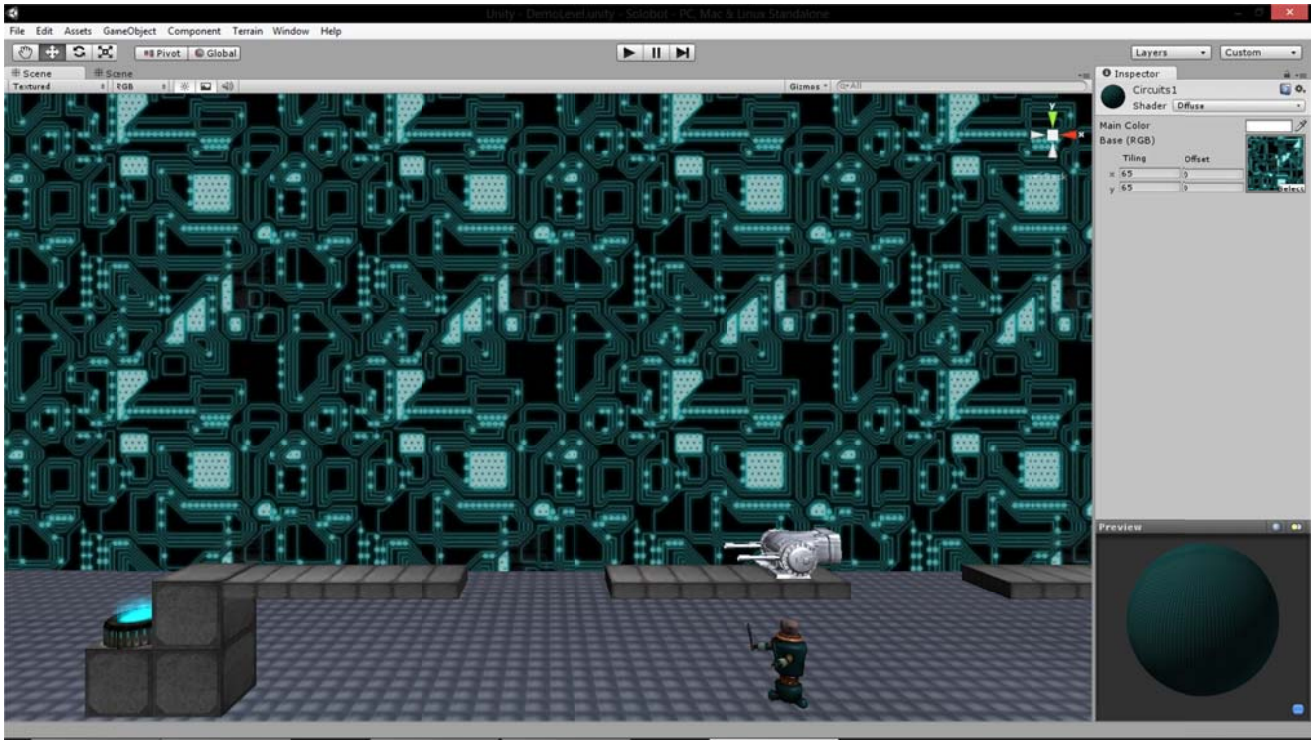
A normal map is an RGB image for which the red, green, and blue channels at a particular pixel represent the normal vector for the surface point at which that texture coordinate is applied in the game. The R, G, and B channels hold a single byte value between 0 and 255 for each color and each value represents the X, Y, and Z coordinates of the normal vector respectively for that particular pixel. When a normal map is applied, the normal vectors from the normal map are used in place of the actual normal vectors from the geometry of the object to which the texture and normal map are applied.

In the case of the crate shown above, instead of the crate only having 6 different normal vectors for which lighting will be varied, using normal mapping there is a different normal for each pixel of the crate shown which is used to calculate the lighting. The following normal map is the one used to create the details shown in the after image above, at each pixel the different RGB values determine the normal vector which will be used in the lighting calculations for that particular pixel on the texture to which the map is applied.

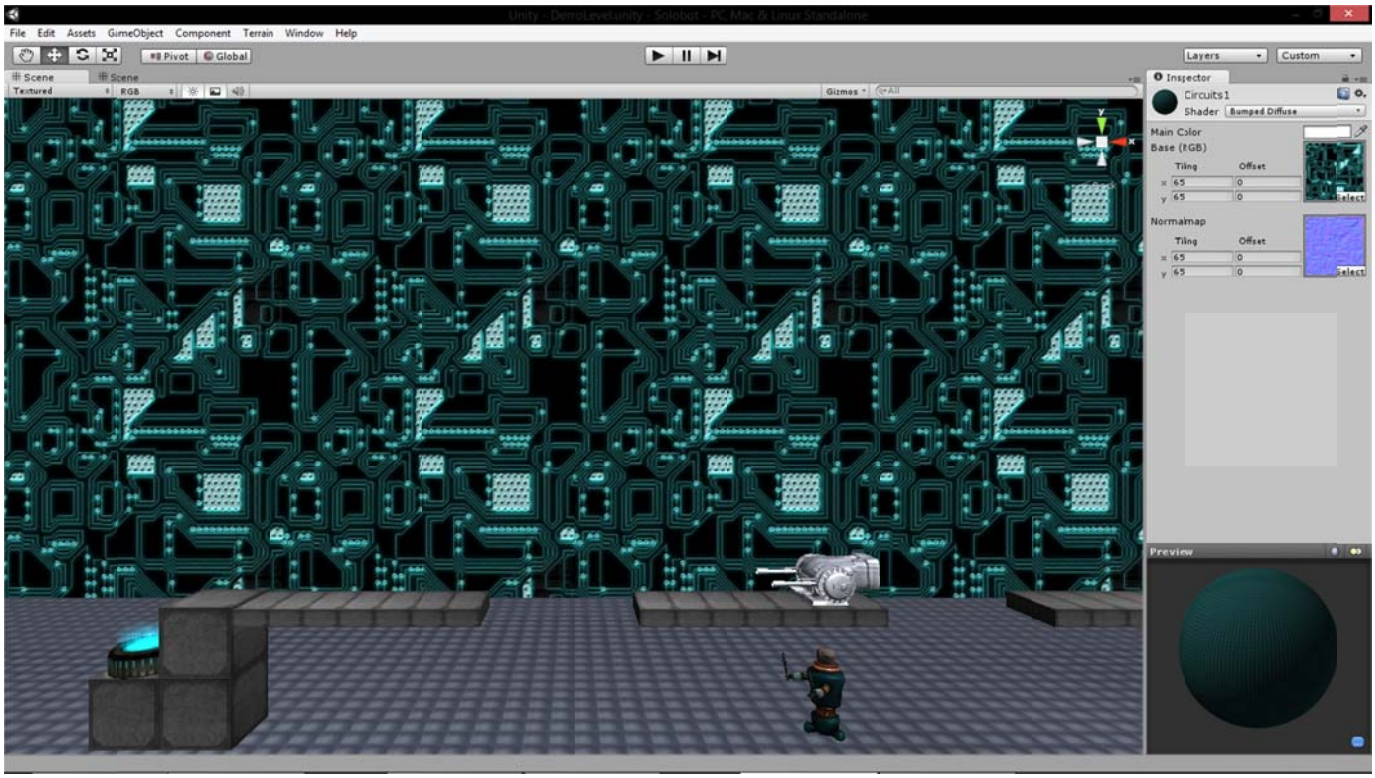




Before Applying Normal Mapping To Our Backdrop



After Applying Normal Mapping To Our Backdrop



I Can Normal Map and You Can Too

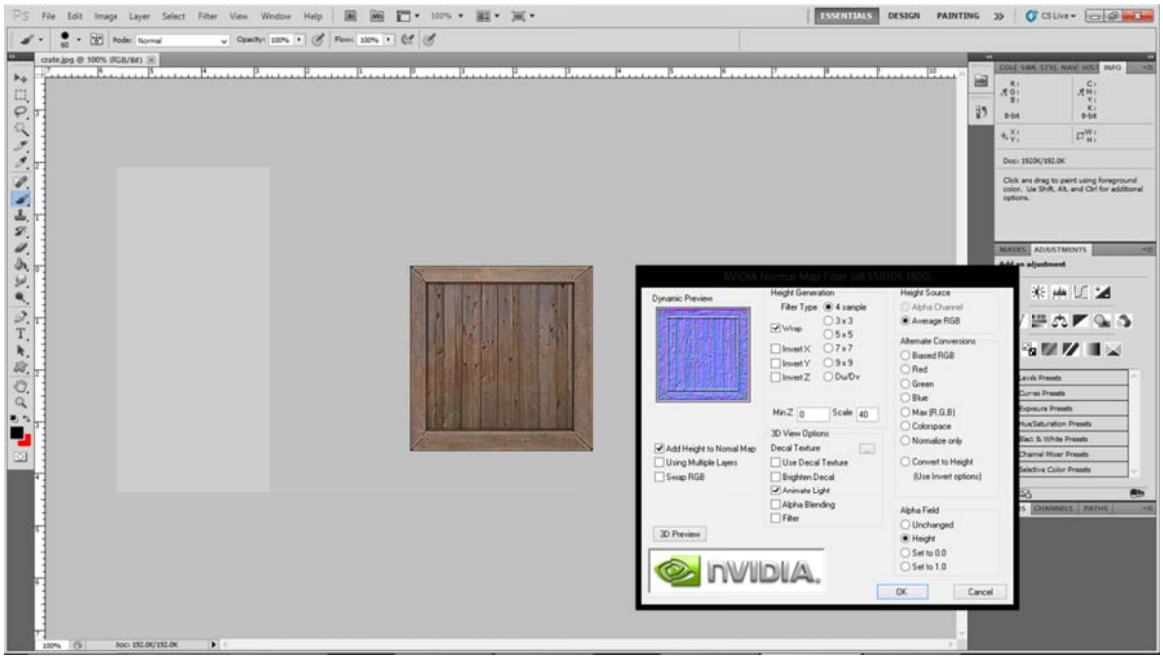
So you may be wondering how you can easily use this technique in games that you create. It would take forever to create a normal map for a given texture by hand, not to mention the effort involved in figuring out what each normal vector should be for each pixel. Instead, usually software tools are employed in order to help generate normal mappings for textures.

NVIDIA's Texture Tools

NVIDIA makes a great filter for Photoshop which allows you to easily create normal maps for your textures. In order to create the normal maps shown above I simply opened them up in Photoshop and ran the normal map filter on it, changed a few values using the (too tiny) preview window, and wa-la- Le normal maps are served.

NVIDIA's Texture Tools for Adobe Photoshop on any version 5.0 or later. This includes all of the creative suite versions as well. It comes with a few extra things, but what you'll be needing is the NVIDIA Normal Map Filter and a working copy of Photoshop.

Once you have that downloaded and installed you can open up a sample texture in Photoshop. I am going to use the wooden crate texture shown at the top of this article. Once you have your texture open, select from the upper menu `Filters --> NVIDIA Tools --> NormalMapFilter` in order to bring up the tool. For the most part it does a good job of generating the normal map pretty well using the default settings but a few you may want to tweak are the **Scale** option which controls the height scale at which the filter is applied and the **Min Z** which controls the minimum height scale.



Solobot Project - Build Instructions – Git Guide / Workflow

Abstract

The source code and workflow will be controlled by Git, written by Linux developer Linus Torvalds, and hosted on the free for open source Git hosting site **GitHub**. The GitHub organization **FreeBeerGames** has been setup for our project. If you haven't been added as part of the dev team for the Github organization then please let email me and let me know so I can give you pull rights for the repo.
The Solobot development repo can be found at **FreeBeerGames/Solobot**. The GitHub repo home page renders the GitHub Flavored Markdown (GFM) in the **README.md file**. This markdown language is very convienient and can result in very simplified HTML documentation. The same README.md file is used on the GitHub Project Page hosted at **freebeergames.github.com/Solobot**. The site located at that URL is simply the gh-pages branch of our repo, we can update it as needed and is actually part of our open source repo.

Git Workflow Overview

Git is a source/version control system which can be downloaded from its **official page**. The Unity IDE is *not* available for Linux, so we will need to develop using Windows. So go ahead and download the latest stable Windows release of the Git utility. The Git utility is your client, you can run it from the Windows command line with the command `git`. There is a **nice cheatsheet** published by the developer's at Heroku that should come in handy for those that have never used Git before. There are a couple of different ways that Git can be used. Since we're using Github and developing on Windows it makes sense to download the **GitHub for Windows client** which will is a GUI built on top of the Git CLI for working with remote GitHub repositories in Windows. Once you have this client installed you have all that you need. If you have not installed the free version of the Unity engine in Windows please browse to Unity3D and download it. Using a Git workflow, we will all be able to have our own development environment, and hopefully everyone will be able to contribute to the project via Git.

Forks vs. Clones (The Movie)

Okay, so most people don't realize this, but forking is actually *not* something that is defined by the Git protocol, and it is added functionality provided by Github with server-side processing. Instead, a git client runs a `clone` command. What this does is basically create a new git repository either in the current working directory or in a given location, pulling the contents of a remote repository so that a local copy of the repo is made. When you create a **fork** on GitHub, what is happening is that GitHub on the server-side is cloning a new remote repository associated with your GitHub account from the remote repository you are forking.

So in the context of our project what I would recommend doing is forking the repository on GitHub, (top right of the repo page). Then on your clone your own fork locally using the **Clone in Windows** button. At this point the Github Windows client should have popped up and downloaded the remote repo.
Even if you do not want to use the GitHub for Windows GUI, I recommend that you download it anyway because it comes with Git Shell, which is a command line shell which allows you to use Linux commands and directory structure on Windows. Regardless of what you use, if you have git installed, after you have forked the main repo **FreeBeerGames/Solobot** you can run the following command in order to clone the repo to the current working directory.

```
git clone https://github.com/FreeBeerGames/Solobot.git
```

You should now right click the repository in the Github client and click **Open Shell Here** and setup our upstream remote repository. You can do this by issuing the following command:

```
git remote add upstream https://github.com/FreeBeerGames/Solobot.git
```

You can then run the following command to view all the remote repositories associated with the local repo:

```
git remote
```

The last command should give you an output of:

```
origin
upstream
```

This indicates that you have two remote repositories associated with the current local repository. The original repository you cloned from. That is your fork unique to your GitHub account. The other remote repository is the upstream repository located at**FreeBeerGames/Solobot**. If you run the following command with the verbose flag you will see that you have both fetch and push rights for your origin repository, however you may only fetch from the upstream repo.

In order to commit changes that you've made locally to your remote fork of the project you should issue the following series of commands:

```
git add .
git commit -a -m "Commit message/what you changed goes here"
git push origin master
```

At this point you will have uploaded your changes to your remote fork of the project. In order to get your changes merged back into the project you submit a pull request which will merge it into the

main repository. You can submit a pull request by browsing to your remote fork on GitHub and clicking the **Pull Request** button torwards the top of the page.

It may be the case, however, that since the last time you forked the main repo we have accepted a pull request and merged changes into the upstream repository. If this is the case, your will have to merge the changes first by pulling the new changes in the upstream remote, then pushing everything once again to your remote fork

```
git pull upstream master
git push origin master
```

You may then complete your pull request.

Bibliography

Cory Gross

http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf

I recently read a presentation released by Valve's Michael Booth that details the AI systems used in the game *Left 4 Dead*. The thriller is set in a post-apocalyptic world overrun with crazed zombies which players must cooperate against in order to survive.

In Michael Booth's presentation he begins by providing some background information for the Left 4 Dead game designed by Valve. The game delivers an intense thriller designed to pit teams of players against swarms of enemy AI's. The level geometry in the games can be very complex and the different AI controlled zombies are of various shapes and sizes. One of the design goals in *Left 4 Dead* was to provide competent "human player proxies" for the AI, giving the player a challenge and making it discernible that the enemy is being controlled algorithmically.

One development of note in the AI system of *Left 4 Dead* is that they originally used the A* path finding algorithm in order to navigate AI controlled characters around the level geometry but they found that this created jagged paths that looked entirely too robotic for crazed zombies. They use a technique called **path optimization** in order to collapse the redundant nodes in the path to create the most minimal and direct path around level geometry and obstacles as possible. This technique will smooth the jagged path created by A*, however, following it directly still looked too robotic for their enemy types. In order to fix this they developed a technique called **Reactive Path Following** which is being used in the game. Using reactive path following, the AI controller moves towards the farthest "look ahead" node on the path that is still visible. This technique combined with trivial local obstacle avoidance algorithms was all that was needed in order to provide realistic path finding for their enemy types.

Having enemy types go after players simply by using the path finding algorithms on the ground was not enough. Players could jump and get to higher ground at which point the enemy AI's could not follow. In order to keep the zombies dangerous they had to develop a path finding algorithm for scaling up geometry. Climbing is also handled algorithmically and is similar to local obstacle avoidance algorithms. The algorithm used can be described as follows

- **Approach**
    - Bot periodically tests for obstacles immediately ahead on its on-ground path using a hull trace.
- **Find Ceiling**
    - If an obstacle is detected another hull trace determines the available vertical space above the bot.
- **Find Ledge**
    - The entire vertical space above the bot is forward scanned by another series of hull traces from lowest to highest in order to find the first unobstructed trace.

    - Another downward hull trace from the first unobstructed trace will give us the exact ledge height.
    - Another series of downward hull traces that step backwards toward the enemy AI will determine the forward edge of the ledge.
- **Perform Climb**
    - Algorithmically pick closest matching animation from dozens of motion captured climb animations at various heights.

John Satriano

http://www.gamasutra.com/view/feature/134566/the_secrets_of_enemy_ai_in_.php?print=1

This article from Gamasutra goes in depth into the enemy AI of Uncharted 2. The article shows all of the variables that the enemies take into account. Emulating human behavior has always been a difficult subject in modern 3D games. Specifically the enemy behaviors which the user must play against. Like an opposing player in chess, the AI must make certain human like responses in order for the player to feel that they are in a realistic situation. The targeting must also reflect normal enemy aim rather than perfect computer aim that is impossible to play against.

The enemies think about a lot of things that a normal player would think about during a firefight: the proximity of the enemy, how many enemies are around you, and what separates you from the enemy. These variables can change whether the AI will charge you, shoot covering fire, or toss a grenade. This allows realistic fighting sequences that immerse the user into a situation where their wits are more valuable than aim and other performance skills. The article gives a complex computational algorithm for the decision making by the enemies and how they work together.

Joshua Kane

http://www.kilobolt.com/game-development-tutorial.html

This website while not very relevant to our project was a valuable source when it came to understanding how to create a game from scratch and more importantly the inner workings of all games. It starts in Java which is a language we're all very familiar with and introduces the game loop and the typical methods it needs to call. It described the details of drawing things to the screen and provides a very detailed tutorial on how to make your own game and walks you through all the steps even how to setup and configure Eclipse. The icing on the cake is that the last lesson then explains how to port games to Android and how to perform game development in Android. It was a valuable resource and taught me a great deal about game design and how video games work and are made.

Carter Michaels

http://www.significant-bits.com/super-mario-bros-3-level-design-lessons

This article describes many of the design decisions that were made in the NES classic Super Mario Bros. 3. It tells about the different things that the player experiences in the first few levels of the game that teaches the player how to play the game without having to hold their hand and tell them explicitly how to play. The author analyzes various aspects of the levels such as powerup placement, the placement of enemies, and the use of rewards to lead the players through the levels. The developers of Super Mario Bros. 3 were able to teach the player every aspect of the game, even the most minute ones, just in those first few levels.

Having played the game myself, I found it very interesting how the developers were able to make even the most absurd concepts, like hitting the side of a wooden block to cause a feather powerup to fly out, seem so organic and simple in Super Mario Bros. 3. One fundamental aspect of the design is that the game teaches the player the fundamentals by presenting them to the player in a way that the player can easily interact with them. When new enemies are introduced, the game puts them in a place where it is easy for the player to avoid or defeat them. When a new powerup is found, there are often clues that allow the player to figure out how to use them easily. The fact that all of this is done without tutorial levels or explicit instructions is a testament to the great game design in Super Mario Bros. 3.

Dillon Daugherty

http://www.gamasutra.com/view/feature/188950/developing_meaningful_player_.php

This article, written by past BioWare developer Alexander M. Freed, describes the design of story elements in games and how they are affected by the player's decisions. He uses specific examples of games with branching decisions affecting the story like Mass Effect, Star Wars: The Old Republic, and Alpha Protocol. The player usually plays these type of games with a specific main character in mind, such as doing all actions 100% good or evil, and while this type of playthrough is valid it often makes it difficult for the player to have a unique, compelling story that they can call their own. Players that think little of their actions and just go with what they think is the "correct" thing to do face little consequences and as a result their story isn't as compelling as one that gets tailored personally to the player.

The author shows that the focus of the game story and decisions should be based on the specific themes of the game and the player's specific actions will reveal themselves on their own. The game should then offer the player difficult decisions that test their values and beliefs, and if they aren't changed they should be pressed. The decisions should be difficult and offer reasons for why a player would change their previous beliefs. Freed gives an excellent example of Star Wars: Knights of the Old Republic, which has an extremely strong turning point that causes the player to question all their previous actions. It provides the player a strong reason to change their player's actions and values, and

for that reason it is a much more compelling and dynamic story than had it not. Games should also acknowledge the decisions the player makes in order for them to be effective. With all these ideas in mind the player's experience with the game story will be much more enjoyable.

William Wallace

http://www.gamasutra.com/view/feature/189899/why_are_we_still_talking_about_.php

In *Why are We Still Talking about LucasArts' Old Adventure Games?,* Frank Cifaldi examines the qualities of old LucasArts games such as Full Throttle and Monkey Island that made them so groundbreaking and memorable. One quality he cites is simply their campiness. LucasArts games often broke the fourth wall and finished their jokes with cheeky grins and nudging, which helped to break any sense of pretention and allow players to feel like they were "in on the joke." This player involvement went a long way toward cementing LucasArts adventure games in the minds of players and industry professionals. Similarly, the gameplay aspects of these experiences were very simple and accessible, which meant the story, characters and jokes were not held back by any learning curve or difficult gameplay sequences which distract from the soul of the game. The third aspect that Cifalid cites as a reason these games were so memorable is the effectiveness with which they hit their target market. These games were designed for and marketed to adolescents, and the cheeky, childish humor and sarcasm of these games hit home with that market. These three qualities serve as an important lesson to game developers who sometimes become distracted by the allure of creating glossy, puffy games which end up without the soul and charm that made classics like Monkey Island so memorable.