# Natural User Interface and Virtual Reality Integration in Video Games

Last Revised: April 1, 2015

Alexandre Wimmers adwimmers@ucdavis.edu
Mario Yepez myepez@ucdavis.edu
Timothy Tong tktong@ucdavis.edu
Valerie Gadjali vgadjali@ucdavis.edu

## 1 Summary

Castle Defense aims to provide the user with a fun and enjoyable gaming experience for everyone. The game makes use of Virtual Reality and a Natural User Interface to provide this experience by using emerging technologies such as the Oculus Rift Dk2 and Leapmotion. Should any problems or questions arise, feel free to contact us at `castledefensegame@gmail.com`

## 2 Resources Required

### 2.1 Hardware Requirements

| | |
|---|---|
| Leap Motion | `https://leapmotion.com/` |
| Oculus Rift DK2 | `https://oculus.com/dk2` |

### 2.2 Software Requirements

| | |
|---|---|
| Leap Motion SDK | 2.2.2.24469 |
| Oculus Rift SDK | 0.4.4 Beta |
| Unity | 4.6.1 |
| Unity Test Tools | 1.5 |

## 3 Build Instructions

1. Checkout the Bitbucket repo: `git@bitbucket.org:ecs193/project.git`
2. Open the project in Unity

3. Optimize the build for the Oculus Rift by going to Prefences→Oculus Vr and checking 'Optimize Builds for Rift'
4. Build the game using the default settings provided by Unity.
5. Run *CastleDefense_DirectToRift.exe.*

## 4 Issue Reporting Procedure

1. File an email report to `castledefensegame@gmail.com` with subject line "[ECS 193 — Issue]".
2. Provide a brief description of the issue.
3. If possible, provide screenshots and steps to reproduce the error.

## 5 Functional Testing

Functional testing will be verified by using automated testing and assertions on class or entity behavior. Unit testing involves the smallest unit possible which tests only the class or entity in question and will mock the behavior of classes it interacts with. Integration testing integrates all entities and creates assertions on specific behavior between the classes without mocked behavior or mocked objects.

### 5.1 Unit Testing

#### 5.1.1 Castle/Player Entity

| Method | Assertion | Description |
|---|---|---|
| Initialize() | CurrentHealth == MaxHealth | Current health initializes with maximum value. |
| LoseHealth($\Delta$) | CurrentHealth == (MaxHealth - $\Delta$) | Current health decreases by a delta amount. |
| Die() | CurrentHealth == 0<br>Verify(LoseHealth(...)).Called(1) | Player health is 0 and that verify that the LoseHealth function was called that caused this death. |
| GameOver() | Verify(Die()).Called(1)<br>GameOver_GUI.alpha == 1 | Verify that the die function was called and that the Game Over GUI becomes opaque. |

### 5.1.2 Enemy

| | | |
|---|---|---|
| Init() | CurrentHealth == MaxHealth<br>Target == Player | Assert that the enemy spawns with max health and that the enemy target is the player. |
| LoseHealth(Δ) | CurrentHealth == (MaxHealth - Δ) | Assert that the castle's health depletes and depletes by a delta amount. |
| Death() | CurrentHealth == 0<br>Speed == 0<br>Verify(DeathAnimation).Called(1)<br>Verify(this).deleted | Assert that the enemy health is 0. Assert that the enemy is no longer moving and that the death animation was called. Then assert that the enemy entity was deleted. |

### 5.1.3 Enemy Manager

| | | |
|---|---|---|
| Init() | SpawnPoints[] != NULL | Assert that the enemy manager has spawn points. |
| Spawn() | CanSpawn == TRUE<br>Verify(SpawnPoint[i].Spawn()).Called(1) | Based on the boolean CanSpawn being true, verify that the spawn point called Spawn() once. |
| Spawn() | CanSpawn == FALSE<br>Verify(SpawnPoint[i].Spawn()).Called(0) | Based on the boolean CanSpawn being false, verify that the spawn point did not call Spawn(). |

#### 5.1.4 Enemy Spawn Point

| | | |
|---|---|---|
| Init() | SpawnType != NULL | Assert that the spawn point has a spawn type. |
| Spawn() | Instance(Enemy) != NULL | Assert that a new instance of enemy was created and is not NULL. |

### 5.2 Integration Testing

#### 5.2.1 Castle/Player Entity

| | | |
|---|---|---|
| GameOver() | ...<br>EnemyManager.canSpawn == FALSE<br>Enemy.speed == 0<br>Enemy.canAttack == 0 | Aside from the same assertions as unit testing, assert that the enemy manager spawn is disabled and that enemies are disabled in both movement and attack abilities. |

#### 5.2.2 Enemy

| | | |
|---|---|---|
| Death() | ...<br>New Player.Score == Old Player.Score + Value | Make the same assertions as unit test. Also assert that the player's score increases by the enemy value. |

## 6 Non-Functional Testing

### 6.1 Security

Not applicable. This product does not contain client sensitive data. The issue of potential hacks into the game is a considerable issue. All Unity games execute through the Unity platform similarly to how Java executes in the Java Virtual Machine. By default, there is "natural" security through this platform. Therefore, if our game was to be compromised, then the issue persists to the entire Unity platform, which falls under the jurisdiction of the Unity Security Team.

### 6.2 Fault Tolerance

Not applicable. The game does not perform any form of persistence.

### 6.3 Performance Testing

Build and run the game. Use the frame-per-second (FPS) counter to poll an average FPS. An average FPS below 30 is unacceptable.