Conducting a Comparative Study of Traditional and Hybrid Xbox Control Systems within a Developed Game

Alexander Poolton

NCCA, Bournemouth University

Abstract:

This paper investigates and analyses the simultaneous use of both the Xbox control-pad and Kinect (a hybrid control system) in comparison to the traditional control system of using only the Xbox control-pad. Both control systems are used within a developed game, which operates as a test bed, and represents that of a typical game found within the mature games genre. Usability tests, conducted on play testers, draw insight into how the hybrid control system compares against the traditional control system. Conclusions are then drawn with regards to to suitability of the hybrid control system, as an alternative to the traditional control system, with emphasis on the mature games genre.

Contents:

1. Introduction …... 7
   1.1 Overview of Kinect-only Games …... 8
   1.2 Overview of Kinect-Assisted Games ….. 11
   1.3 Problems Posed by the Kinect …... 12

2. Previous Works …... 13
   2.1 Nintendo Wii Remote and Nunchuk ….. 13
   2.2 PlayStation Move Wand and Move Navigation …................................ 14
   2.3 Hybrid Control System Proposal …... 16

3. Technical Background …... 17
   3.1 Kinect Development …... 17
   3.2 NITE Skeletal Tracking …... 18
   3.3 XInput …... 19
   3.4 Development Environment …... 19

4. Stages of Implementation …... 21
   4.1 Design Model and Usability Testing …... 21
   4.2 Game Outline …... 21
   4.3 Hybrid Control System …... 22
   4.4 Game Mechanics …... 26
      4.4.1 Player Behaviour …... 26
      4.4.2 Flash Light …... 31
      4.4.3 Gun and Bullet Effects …... 31
      4.4.4 Enemy Behaviour …... 34
      4.4.5 Door and Key System …... 36
      4.4.6 Room Event System …... 37
      4.4.7 Audio Events ….. 38
      4.4.8 Light Events …... 39
      4.4.9 Heads-up Display …... 39
      4.4.10 Re-playability Factor …... 40
      4.4.11 Game Settings and Main Menu …................................... 41

5. Results of Usability Tests …... 44

6. Conclusion ….. 46

7. References ….. 48

8. Bibliography …... 51

Appendices …... 52

List of Figures:

Figure 1: An image illustrating a user operating the Xbox 360 control-pad.

Figure 2: A picture illustrating the mini-game “Rally Ball”. “Rally Ball” is one of a series of mini-games which make up the Kinect Adventures Xbox game. Note, the player's limited movement space within the game.

Figure 3: An illustration showing a player performing a gesture, while using the Kinect.

Figure 4: A picture illustrating a Kinect side-task/mini-game within the Harry Potter and The Deathly Hallows game.

Figure 5: An image illustrating the Wii Remote (right) and the Wii Nunchuk (left) being held by a player.

Figure 6: An image illustrating the PlayStation Move Wand (right) and PlayStation Navigation controller (left) held by a player.

Figure 7: An image illustrating two successful mature games (first-person shooters) using the same outlined control configuration (discussed previously).

Figure 8: A diagram illustrating the body points that NITE tracks.

Figure 9: An image illustrating the OpenNI pre-built skeletal rig. Note, red dots to indicate the tracking points have been included.

Figure 10: An image illustrating the two Kinect points for either hand. Only one set, a Kinect point and the player's dominate hand will be used at any given time in the game.

Figure 11: A digram illustrating the Kinect point and Kinect hand tracking point retaliative to each other.

Figure 12: A set of 3 images illustrating the Kinect collision box, with additional collision boxes placed on the hands to collect collision between the boxes.

Figure 13: An image illustrating the object identification function. Note, the object's name printed to the HUD in the lower-left hand corner.

Figure 14: A diagram illustrating the inheritance structure of the inventory items used

within the game.

Figure 15: This image illustrates the gun inventory item being added to the player's inventory. The player can toggle the inventory GUI window on and off, by pressing a button on the control-pad.

Figure 16: An image illustrating a “light crate” being lifted within the game.

Figure 17: An image illustrating the 3 different types of pick-up within the game (battery, ammunition and health).

Figure 18: An image illustrating the player's different hit areas. Each hit area is a box collider, and detect hits from enemy attacks.

Figure 19: An image illustrating the flash light turned on, within an area of the game where the lights are turned off. Note, the flash light power level displayed on the left-hand side of the screen.

Figure 20: An image illustrating the addForce method, adding a velocity vector to the rigid-body of an object, from the bullet that struck it.

Figure 21: An image illustrating a bullet striking the surrounding environment, and instantiating a bullet hole and sparks.

Figure 22: An image illustrating a behaviour of a bullet striking an enemy.

Figure 23: An image illustrating the orc's box collision hit areas. The hit areas determine where the player's bullet hit, and deals damage to the orc accordingly.

Figure 24: A diagram illustrating the orc's firing system, showing the original line cast destination, and the ultimate destination (with the random numbers added to the destination’s Cartesian co-ordinates).

Figure 25: A image illustrating a key-card and door within the game.

Figure 26: An image illustrating a puzzle room (left) and an enemy room (right).

Figure 27: An image illustrating the player's HUD, in which the three HUD elements (flash light, ammo and health statistics) have been highlighted.

Figure 28: An image illustrating the end of game statistics screen, informing the player of their performance throughout the game.Figure 29: An image to illustrate the secret collectable items which are hidden throughout the game.

Figure 30: An image illustrating the main menu of the game. The image shows the different sub-menus the player can open, and explore.

Figure 31: An image illustrating the options GUI (a sub-menu of the main menu). The player can use the GUI elements (sliders and a toggle) to modify the game's settings.

Figure 32: A graph illustrating the number of times a player died, within both versions of the game

Figure 33: A graph illustrating the time taken to by a player to complete the game (the time for both versions of the game have been recorded).

1. Introduction:

The purpose of this project is to develop a game which shall compare and critically analyse two controls systems for the Xbox 360 platform, with emphasis on the mature games genre. The two control systems are the Xbox 360 control-pad (only) and a hybrid control system (the Xbox 360 control-pad and Microsoft Kinect simultaneously being used by the player). The developed game will encompass as a series of usability tests, to test these control systems against each other. From these usability tests, results can derived with regards to how effective the hybrid control system (simultaneous use of the Xbox 360 control-pad and the Microsoft Kinect) performed, in comparison to the traditional configuration, of only using the Xbox 360 control-pad.

The two pieces of technology (Xbox 360 control-pad and the Microsoft Kinect) are both used to facilitate Human-Computer Interaction between the player and the Xbox 360 platform. However, both achieve interaction, in a manner distinct of each other, and are subsequently targeted at different user groups.

The Xbox 360 control-pad, the principle HCI hardware for the Xbox 360 platform is a variant of the traditional "game-pad" (as used in previous gaming platforms) and as such it is employed across all genre of games, developed for the Xbox 360 platform. The controller itself is designed to used by the player with both hands holding either side, whilst operating a combination of joysticks and buttons. The design itself is a refinement of many iterations of traditional “game-pads” which have been proven (repeatedly) as the most successful HCI input device to date, for gaming platforms. Thus, users are instantly familiar with how to use these types of HCI controller devices, in order to successfully operate games across multiple different genres.

*Figure 1: An image illustrating a user operating the Xbox 360 control-pad (*Muich, 2006).

The Kinect device is a recently developed piece of HCI hardware (released November, 2010) which tracks a player's body movement through an infra-red camera. Since the Kinect's inception, games developed for it's use have fallen into two categories: games in which the Kinect is the game's primary and only controller, and games where the Kinect acts as the game's secondary controller (the primary controller being Xbox 360 control-pad).

1.1 Overview of Kinect-only Games:

Games which employ the Kinect as it's primary (and only) control device, are typically targeted at "casual" players, and are thus heavily "mini-game" orientated. These games typically provide players with a series of isolated tasks, in which they will have to perform a series of body movements or sounds (using the Kinect's microphone) in order to complete the set of tasks at hand. For instance, "Kinect Adventures" (Microsoft Games Studio, 2010) presents a player with a mini-game called “Rally Ball” (see Figure 2) in which the player must move their limbs in order to bounce balls at a formation of blocks (a variation of a classic arcade game).

Figure 2: A picture illustrating the mini-game "Rally Ball". "Rally Ball" is one of a series of mini-games which make up the Kinect Adventures Xbox game. Note, the player's limited movement space within the game (Henry, 2011).

Typically, these mini-games are complied together to create a compendium, to create an overall game. For instance, "Kinect Adventures" illustrates a typical example of the "mini-game compendium" genre where several mini-games are brought together for player's to play against the computer AI or other human players. Thus, these games are typically targeted at "casual" players, and formulate the overwhelming majority of "Kinect-only" games on the market today. Which in turn has caused Kinect-only games, to be rejected by more "serious" gamers, and thus these games have have failed to penetrate the mature gaming genres, such as first-person shooters or role playing games. Additionally, Kinect only games, which are not apart of the mini-game compendium genre, are generally sports games which feature the same set of tasks a typical mini-game would feature. For instance, the tennis game "Virtua Tennis 4" (Sega, 2011) is in reality a refined version (as the player's feet remain in a static position in the same manner) of the "Rally Ball" mini-game as seen in "Kinect Adventures" (see Figure 2).

Coupled with the mini-game driven nature of Kinect-only games, another major deterrent for games developers to develop games for "serious" gamers, are the limited set of controls the Kinect devices offers, toward the more mature genre of games (such as first-person shooters or role-playing games). This is apparent in the fact that, whilst the Kinect tracks a player's body movements, problems can occur when a player accidentally covers a limb (such as a hand) which causes the Kinect to lose tracking of

that limb. Which given the precision and accuracy required for a player to operate a first-person shooter (in comparison to other gaming genres) would only cause frustration and poor usability (in terms of Human-Computer Interaction) among serious gamers. Furthermore, the Kinect is designed “to be operated between a distance of two and ten metres” (Microsoft, 2010) and thus enabling players to walk back/forward, left/right and rotate to a high level of accuracy (which is a requirement of first-person shooters) would again cause poor usability (i.e. moving closer and further away from the television screen would prove unsuitable). Thus, creating a more mature Kinect-only game is not in keeping with what the Kinect was designed for (namely for a player to stand in a somewhat static space, while performing body movements and gestures, facing toward the Kinect device – see Figure 3).

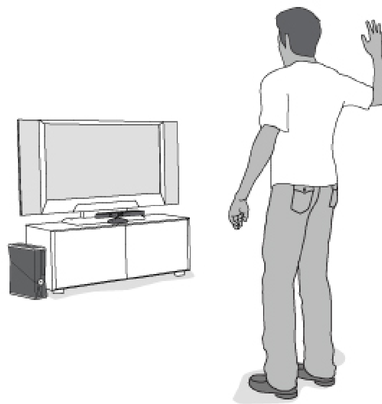


Figure 3: An illustration showing a player performing a gesture, while using the Kinect (Brutal Gamer, 2010).

Additionally, the level of input controls required for a first-person shooter or role-playing game is typically quite substantial, in comparison to other game genres. For instance a typical shooter might require at least 6 to 8 essentially actions, such as: jump, fire, reload, throw grenade, open door etc. to successfully operate the game. Whereas some of these actions (for instance actions such as opening a door) can be gesture based (i.e. the player pushing their hand forward), the number of unique gestures and/or actions are limited, and much less accessible to a player, in comparison to the familiarity found when using a traditional controller. Furthermore, the player could in theory, perform movement controls (forward, back, left, right) with one hand, and look controls (look up, down, left, right) with the other hand (i.e. moving each individual hand independently to control both look and move functions). However, in doing so, the player would then be unable (or at least very limited) to perform several

actions at once, and may possibly experience some interference between commands (i.e. trying to walk forward, turn and shoot simultaneously).

1.2 Overview of Kinect-Assisted Games:

The other category of games which utilise the Kinect, are games in which the Kinect is used as a secondary controller. Games which are developed in this manner, use the Kinect as a "side-task" tool, where the player uses the Xbox 360 control-pad for the majority of the game, and is then prompted to place the traditional Xbox 360 control-pad down, and switch to using the Kinect. Once the Kinect task is completed the player then resumes playing with the Xbox 360 control-pad. For instance, in the "Harry Potter and The Deathly Hallows – Part 1" (EA Bright Light, 2010) game, players perform gestures to launch spells at enemies (or other various actions, such as throw potions, or hide under a cloak) as a part of an "on-rails" series of challenges in either single or co-operative mode (see Figure 4). However, when playing the main story mode of the game, which consists of a third-person character control, as seen in games such as "Grand Theft Auto IV" (Rockstar North, 2008) players use the Xbox 360 control-pad. This methodology of using different control devices for different tasks again illustrates, how the Kinect isolated by itself, cannot deliverer the level of control required to operate a detailed set of controls. Furthermore, this change between control devices can cause the game to appear somewhat disjointed, and in turn cause the Kinect device to be perceived as a "gimmick" or cumbersome by the mature gaming community.

Figure 4: A picture illustrating a Kinect side-task/mini-game within the Harry Potter and The Deathly Hallows – Part 1 game (IGN, 2011).

1.3 Problems Posed by the Kinect:

To reiterate the two major problems the Kinect itself poses when developing games for a mature audience are):

- The Kinect alone does not provide a high enough level of input/interaction to successfully operate a game targeted at mature players, such as a first-person shooter or role-playing games. Thus, these games cannot be developed as "Kinect-only".

- Forcing the player to swap between the Xbox 360 game-pad and the Kinect causes the player's experience of a game to feel disjointed, and thus provides a poor user experience. Furthermore, it re-enforces the idea that the Kinect is simply an add-on tool, and not integral to the overall game.

Thus, this thesis will now discuss solutions, which will address the problems, that have been outlined (see above).

2. Previous Work:

Currently there are two configurations of HCI devices on gaming platforms, which combine two HCI devices to enable the user to control: both their movement and look functions, whilst also providing a suitable level of input controls for player “actions” (i.e. jump, use, fire weapon etc.). These three criteria encapsulate the key functionality, typical mature games require. The two HCI device configurations are the Nintendo's “Wii Remote and Wii Nunchuk” (Nintendo, 2006) for the “Nintendo Wii” games console and Sony's “PlayStation Move Wand and PlayStation Move Navigation Controller” (Sony, 2010) for the “Sony PlayStation 3” games console.

2.1 Nintendo Wii Remote and Nunchuk:

Nintendo's Wii Remote and Wii Nunchuk configuration is operated by the player holding each HCI controller in opposing hands (the Wii Remote, being the primary controller, is held in the player's dominant hand). The “Wii Remote” enables the player's position to be calculated (using trigonometry) via an infra-red light sensor bar, that is positioned level to the display. The Wii Remote itself consists of 11 input buttons, and a built-in accelerometer, which is typically used to calculate the rotation of the controller, or exerted force of the player's hand (i.e. the punching force of a player, within a boxing game).

Figure 5: An image illustrating the Wii Remote (right) and the Wii Nunchuk (left) being held by a player (VaroLogic Blog, 2008).

The Wii Nunchuk, being the Wii's secondary controller, is held in the player's weaker hand and is connected via cable to the bottom of the Wii Remote (see Figure 5, above). The Wii Nunchuk provides the player with a joystick, and two additional input buttons. A typical configuration established within the majority of mature Wii games, is for the Wii Remote to control the player's in-game movement (up, down, left, right), while the Wii Remote provides the player with the look/aiming functionality (by controlling the player's view within the game). The hardware itself promotes this methodology, by providing the player with a button located on the underside of the Wii Remote, which imitates that of a gun trigger.

2.2 PlayStation Move Wand and PlayStation Move Navigation:

The PlayStation Move Wand and PlayStation Move Navigation controller, use exactly the same configuration as established by the Nintendo Wii. Where the Move Wand is placed in the player's dominant hand, to control the player's viewpoint and the Move Navigation controller is used control the player's movement within the game. However, the technology does differ, in the way it determines the player's hand position. The PlayStation Move instead utilises a similar method to the Kinect, by using a stationary camera the "PlayStation Eye" to track the distinctive orb positioned on top of the Move Wand (see Figure 6, below), thus tracking the player's hand position. The Move Navigation controller also imitates the Nintendo's Nunchuk, and provides 8 inputs buttons for the user.

Figure 6: An image illustrating the PlayStation Move Wand (right) and PlayStation Navigation controller (left) held by a player (Kotaku, 2011).

Upon investigating both of these HCI control configurations, the most striking implications are their similarities and not their differences (which are negligible). This key point highly suggests that this configuration of providing a player with two HCI devices in opposing hands. Where one is able to control the player's viewpoint, and the other the player's movement, in which the dominant and weaker hands control each respectively, is the best solution to the problems posed (see Section 1.3 - Problems Posed by the Kinect). The reasoning for this configuration (of one controller typically being used in the dominant and weaker hands) is due to the fact that the look/aim functionality requires a higher degree of accuracy in comparison to the move/navigation functionality. Thus, the HCI control device that controls the player's viewpoint/aim is typically placed in their dominant hand. Furthermore, to re-enforce this argument, both configurations scrutinized (see above) have been used within an array of successful mature games. Games such as "Red Steel" (Ubisoft Paris, 2006) and "Killzone 3" (Guerrilla Games, 2011) developed for the Wii and PlayStation 3 respectively, both illustrate that the discussed control configuration works well with respects to real-world use. Furthermore, illustrating that this control system appeals to "serious" gamers (addressing the outlined problem of the Kinect-only games, not being able to penetrate the mature games genre, see Section 1.3 - Problems Posed by the Kinect).

Figure 7: An image illustrating two successful mature games (first-person shooters) using the same outlined control configuration (discussed previously) (Syfan Media, IGN, 2011).

2.3 Hybrid Control System Proposal

Therefore, the research conducted, suggests the solution to the two key issues posed in the introduction of this thesis (see Section 1.3 - Problems Posed by the Kinect) is to combine the two Xbox HCI devices together (the Xbox 360 control-pad and Kinect) in a “hybrid control system”.

Thus, from the research gathered, the player would use their dominant hand to control their viewpoint within the game. This function could be controlled either with the Xbox game-pad or Kinect. However, the advantage of having a freely moving hand, over another hand holding one side of a controller (in which it was not designed to be held) would favour the hand being tracked via the Kinect, to control the player's viewpoint. Thus, with the player's “Kinect hand” the player would able to control their viewpoint in the relation to where they moved their hand (i.e. if the player moves their hand upwards, then the in-game camera also looks upwards).

Gestures could also be used by the player's “Kinect hand”. However, the movement would have to be limited in order not to create “interference” between desired commands (i.e. the player waving their hand to open a door, would cause them to look left and right with each wave, which is not a desired effect). Thus, gestures should be limited to the z-axis or the hand (thus not affecting the x or y axis). For instance, gestures, such as the player reaching out (away from their body) to grab an item, or to push a door open.

Subsequently, the Xbox 360 control-pad would be placed in the player's weaker hand, providing a joystick to control the player's movement (up, down, left, right) and 8 buttons input buttons. This number of input buttons, would provide the player with a suitable number of inputs in order to play a mature game, addressing problem outlined previously (see Section 1.3 - Problems Posed by the Kinect).

3. Technical Background

In order to test the proposed solution, a game will be developed, which will implement: the move and look functionality previously outlined (see Section 2.3 – Hybrid Control System Proposal), several Kinect related functions (i.e. pick-up item, open door) and other common functionality found within a game (i.e. enemy artificial intelligence, goal-orientated game play and player feedback).

The purposed game being developed, shall be a first-person shooting game, as this is a mature genre, it correlates to one of the outlined problems to be addressed (see Section 1.3 - Problems Posed by the Kinect). Furthermore, given the constrained time frame for implementation, a role-playing game (which has been previously discussed) would be extremely difficult to develop as games in this genre is very “content-heavy”, and would add little to the results of this thesis. However, elements (such as a picking up items, and a player inventory) shall be added, to provide adequate insight into how the usability relates to both the first-person shooter and role-playing genre (the most popular mature gaming genres).

3.1 Kinect Development:

As apart of the HCI to control the player's look functionality, the Kinect will be used to track the player's dominate hand. The Kinect currently has three prominent developing platforms: OpenKinect, Microsoft Kinect SDK and the OpenNI framework coupled with NITE middleware.

OpenKinect is an open-source framework, which provides users with direct access to the Kinect's depth information and camera. However, skeletal algorithms to track the player's body are not currently supported, and implementing the algorithms from scratch is outside the scope of this thesis.

The Microsoft Kinect SDK supports skeletal algorithms, however the SDK itself is still in beta stages (as of time of writing) and thus has not had time to mature as a piece of software. Additionally, the Microsoft SDK does not have any official or unofficial game engine support, due to it still being in it's infancy.

The OpenNI framework plus NITE middleware is a mature Natural Interaction Device development platform. The framework itself supports numerous Natural Interaction devices including the Kinect and Asus' recently announced "Wavi Xtion" (Asus, 2011) motion sensing system. Furthermore, the OpenNI framework + NITE middleware currently supports two game development engines, ORGE and Unity (in which it has created wrappers for both).

Thus, given the maturity of the OpenNI framework plus NITE middleware, it's support for multiple types of Natural Interaction devices (enabling more flexibility and future expansion of the implementation) and it's official support of two current game engines; it is the most suitable choice to develop with, during this project.

3.2 NITE Skeletal Tracking:

NITE (the middleware placed on top of the OpenNI framework) provides the skeletal tracking algorithms used to track several points on the player's body by the Kinect. In total NITE tracks up to 14 points across the user's body, as shown in Figure 8 (see below). However, as a part of this project, the only points that are significant, are the player's dominate hand (the hand they will control their in-game view with). Therefore, before the game starts the player will select which hand this wish to control their view with (i.e. is the player right or left handed), and in turn the game will use the specified dominate hand to control the player's view. Additionally, the head transform point may also be used to roll the camera, dependant on the rotation of the player's head, replicating a lean function found within many first-person shooting games.

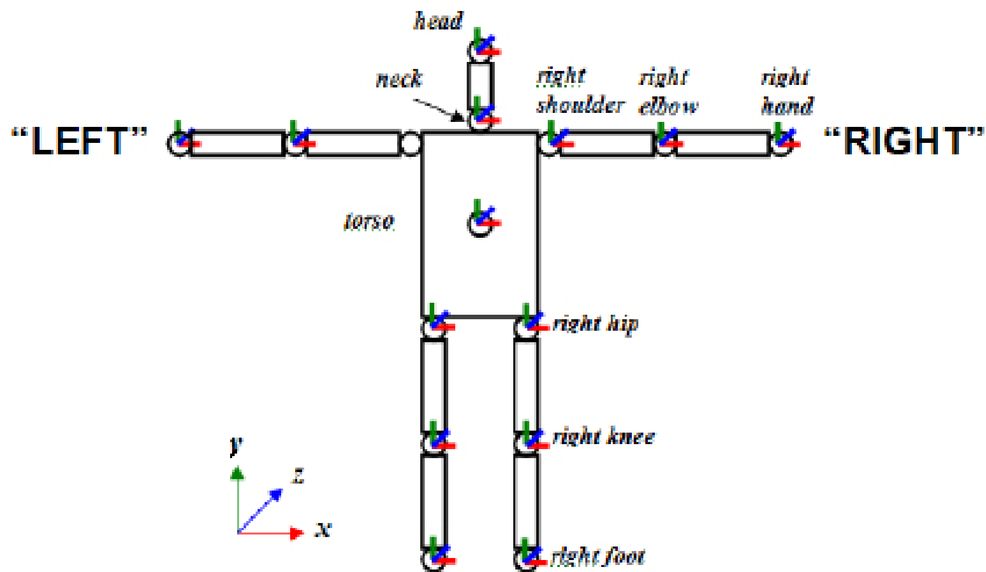


Figure 8: A diagram illustrating the body points that NITE track (Tribal Lab, 2011).

3.3 XInput:

XInput is a Microsoft API for interfacing with input devices namely the Xbox 360 controller. The system provides developers with button mapping (thus functions can be mapped to various buttons and axis/joystick controls). Therefore, allowing developers to re-map the controller to utilise the functionality within their developed game. XInput is supported by both ORGE and Unity. Note, XInput is supported by numerous game engines. However, as ORGE and Unity are the only game engines supported by OpenNI, they are more suitable options to develop the game with.

3.4 Development Environment:

Utilising an existing game engine, in favour of creating a small game engine from scratch was chosen, due to the time constraints of the project; coupled with the fact the focus of this thesis, is a comparative study of Xbox 360 controls, and not to develop low-level game technology. Therefore, allowing the project to focus on the HCI principles, usability and integration of controls within a developed game. As opposed to additionally creating a basic game engine to test these aspects, was deemed the most suitable approach. Furthermore, using an existing game engine, would allow for a game with a much wider scope and larger set of game mechanics. Which in turn would provide a more diverse set of functionality to test the control configuration (especially

the Kinect itself). This would furthermore provide richer usability results and a much closer (although still extremely small in comparison) resemblance to a fully featured commercial game (allowing the results to have some tangible meaning to game developers).

The chosen development platform for this project will therefore be the Unity game engine. The platform was chosen due to it's support of OpenNI and XInput. Furthermore, the Unity game engine has various components, such as rigid body dynamics and ray casting which are built into the game engine, enabling developers to create behaviours and functionality using them. Therefore, increased efforts can be placed focusing on key principles of the implementation (i.e. creating a game which acts as rich usability testing environment) which in turn will provide richer usability results to draw conclusions from.

4. Implementation Stages

4.1 Design Model and Usability Testing:

Once a suitable development environment had been chosen, a development model was selected, to ensure the implementation of the game was performed in an effective manner. The iterative development model was chosen, as it is a common practice within the games industry to ensure games under development receive iterative usability tests (play testing). The purpose of iterative testing is to provide developers with constant feedback, on key principles of the game, such as usability, bug-testing and general user feedback. Thus, iterative usability tests shall be carried out regularly (weekly) throughout the game's development. The group of play testers shall consist of a mixed age group, with differing gaming experience (i.e. some inexperienced with games, while others being very experienced with games). This mixed demographic will provide a broader horizon of usability test results, and thus richer insights into whether the problems outlined (see Section 1.3 - Problems Posed by the Kinect) were addressed. The practice of "using a mixed demographic of test subjects, is a common practice within usability testing, and is known as Hallway Testing" (Rubin, Chisnell, 2008).

Once formed, the usability test group consisted of 8 mixed participants (each with different levels of gaming experience). Each usability test would consist of a play test of a specific game mechanic, while qualitative studies (the user would be asked a series of questions, i.e. "Do the controls feel easy to operate?") were conducted. The game mechanic would then be modified or kept the same, dependant on the usability group's feedback.

4.2 Game Outline:

The game itself follows a linear goal-orientated gameplay style, in addition to a simple linear story arc. The game is set within a prison wing, where two prison guards are currently working. The game starts with the main protagonist (one of the prison guards, the character, the player plays as during the game) carrying out a routine patrol of the prison wing. Upon the discovery of blood, the player is lead to discover the second guard in a pool of his own blood. The player then discovers the inmates of the prison

(orcs) have escaped, and have entrenched themselves throughout the complex. The player must then fight their way out of the complex, solving various puzzles, and ensuing gunfights, in order to survive.

4.3 Hybrid Control System:

The hybrid control system consists of the Xbox control-pad (to move the player) and the Kinect control system (which enables the player to look, and perform various other actions, such as open doors and pick up objects).

As previously stated the Unity engine uses XInput (see Section 3.3 - XInput), and thus interfacing with the Xbox controls is already integrated within the Unity engine. Therefore, all that is required to use the Xbox 360 control-pad within Unity is to re-map the buttons and joystick axis to their desired functions. Furthermore, Unity offers a pre-built first-person control system implemented within the engine, as a part of it's standard assets. Thus, instead of recreating a standard first-person control system, it was decided it was better to re-use the functionality and focus efforts on building the Kinect control system, and the rest of the game. This is largely due to the fact that building a standard first-person control class from scratch, would be a redundant effort, as it has been developed countless times before within the industry. Once the Xbox axis controls (the joysticks) had been mapped to the first-person control class (to enable the in-game camera to be moved forward, backwards, left and right) the Kinect control system needed to be developed.

One of the key reasons for using Unity is that OpenNI officially supports the game engine, and has created a wrapper to use NITE functionality within Unity. Furthermore, a sample project is included with the wrapper, which encompasses a pre-built skeleton (as shown in Figure 9, see below). After testing the OpenNI rig with the Kinect, optimizations were made by disabling non-essential features, such as displaying the users depth image within game. With the default features OpenNI used, the game was ran at 50 frames per second. However, after optimization (disabling of non-essential features) the game ran at 160 to 180 frames per second.

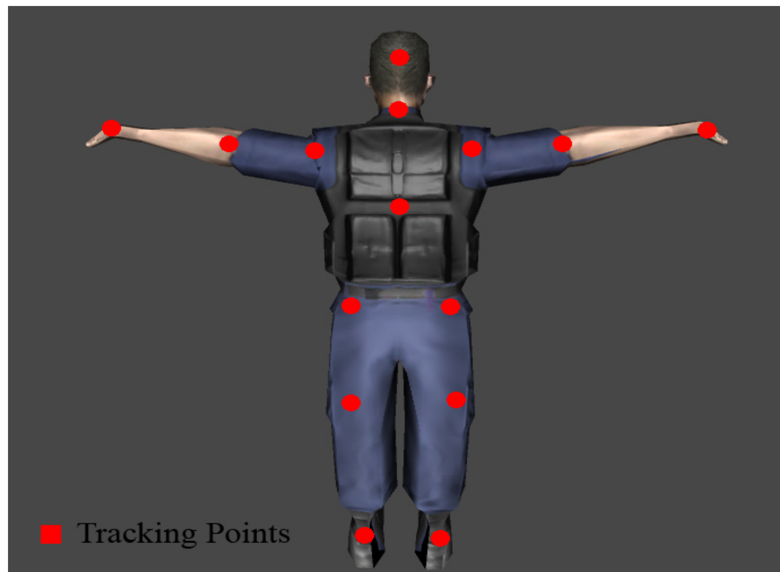


Figure 9: An image illustrating the OpenNI pre-built skeletal rig. Note, red dots to indicate the tracking points have been included.

To enable the player to control the first-person camera within the game, in relation to where they moved their dominate (or "Kinect hand"), a point of reference was needed. This point of reference would be used to calculate the vector magnitude between the hand transform and the point of reference. From this vector magnitude, the camera' pitch and yaw could be rotated according, dependant on if the player's hand moved up or down in relation to the point of reference. The point of reference which was entitled the "Kinect point", was placed according to a series of usability studies, investigating where users felt most comfortable holding their hand (in front of their torsos) for a long period of time. From this usability study it was found that users tend to rest their elbow just above their hips, in order to support it, when holding their arm out for lengthy periods of time. Therefore, providing even though this thesis will present a prototype game (which will last less than 15 minutes) it should be still kept in mind that commercial games typically encompass a playing time of 20 hours of more, and thus the player will want to be positioned in the most comfortable position possible. Thus, from this usability study the "Kinect point" was positioned just under the player's shoulder of their dominate hand (see, Figure 10). Obviously, there required to be two Kinect reference points, one for each hand (dependant on which the player selected as their dominate hand). Thus, according to which hand the player select to use to control the in-game camera, the opposing Kinect point will be disregarded and the Kinect point of the corresponding hand will be used.

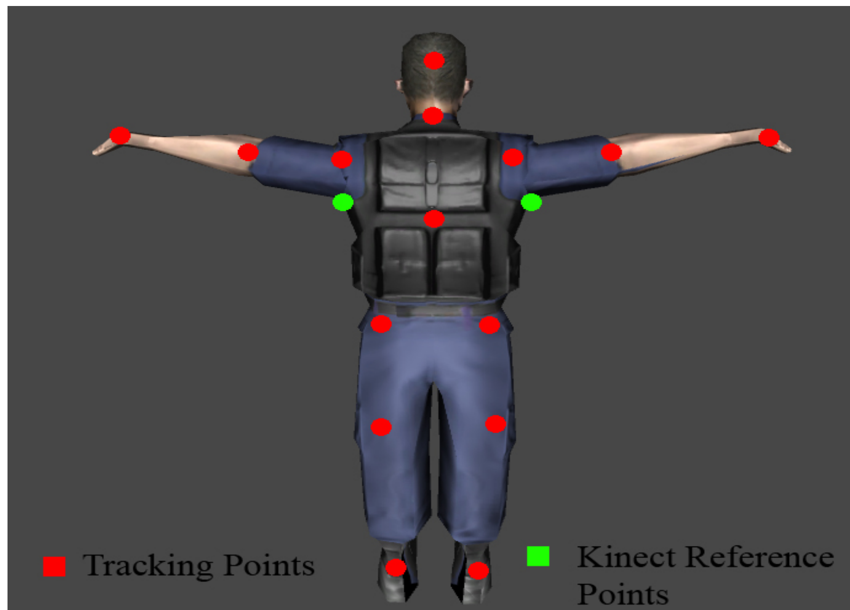


Figure 10: An image illustrating the two Kinect points for either hand. Only one set, a Kinect point and the player's dominate hand will be used at any given time in the game.

The displacement between the hand point and the Kinect point is calculated between a scale of -1 and 1 (as shown in Figure 11, see below) and then multiplied by a scaling factor (sensitivity); which allows the player to look/aim at a faster or slower rate (a function found within the majority of first-person shooting games). When the player's hand is at it's natural rest point (the Kinect point) the camera remains at a fixed position. Thus, enabling the player to change the viewpoint of the camera when they desire (by moving their hand, in the desired direction).

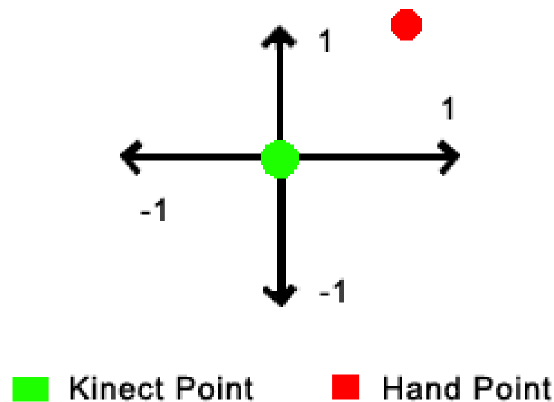


Figure 11: A digram illustrating the Kinect point and Kinect hand tracking point retaliative to each other.

The final component of the Kinect control system was to implement the “reach” functionality (which would later be used to invoke the “use”, fire gun and pick up actions). The reach functionality acts as an extra input button for the player, by enabling them to invoke an action (i.e. firing their gun or opening a door) by reaching out in front of their torso. This functionality was initially achieved by using the z-axis magnitude between the Kinect point and the Kinect hand point. For instance, once the hand exceeded a pre-set threshold (i.e. it was far enough away from the user's body) a boolean value would be set to true, enabling the player to pick up an item. However, upon experimenting another method provided a “crisper” and more satisfactory response, as stated by users, as part of the usability tests. This method was to parent a collision detection box to the player object within Unity and pre-set the distance (similar to the threshold value, implemented in the previous method) between the torso and collision box (see Figure 12, below). The same boolean value (mentioned earlier) was then set to true or false, dependant on if the collision boxes placed on the pre-defined (as selected by the user, at the start of the game) dominate hand of the Kinect rig, had entered or exited the Kinect rig collision box (the collision box place in front of the rig). The pre-set distance between the rig and collision box was tested with users in the testing group, with both shorter and longer arms, and provided the correct results without failure. Thus, based on the user feedback and the perfect test results, this method was kept, as opposed to the former implemented method of implementing the pick up/use functionality.

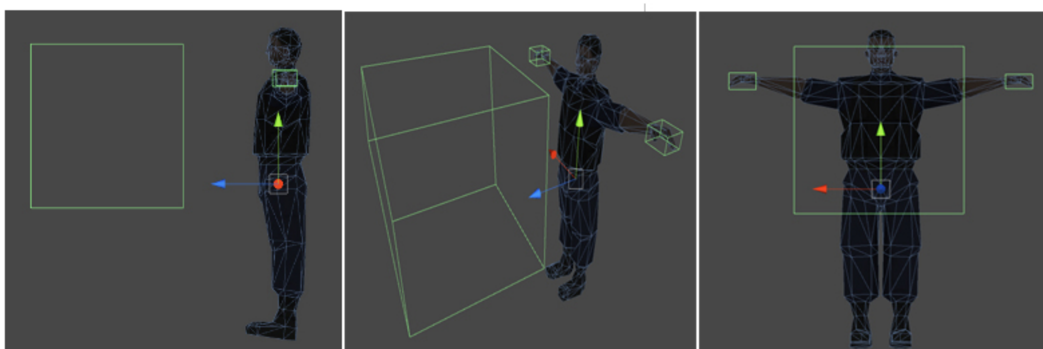


Figure 12: A set of 3 images illustrating the Kinect collision box, with additional collision boxes placed on the hands to collect collision between the boxes.

The player's “lean” functionality, which allows players to lean in-game (around corners and objects), was also implemented. Players can lean in-game, by leaning left or right, in front of the Kinect (or simply rotating their head). The direction the player leans (or rotates their head) will cause the in-game camera to also rotate/lean in the same direction.

4.4: Game Mechanics:

Upon completion of the Kinect control system (which enabled the player to freely move and look) a game in which to test the control system needed to be developed. The development of the game was broken down into manageable components, each implementing a different game mechanic. The game mechanics chosen to be implemented, are those that are typically seen in first-person shooters or role-playing games, such as: a weapon system, enemy artificial intelligence, door and key systems, pick ups, event-driven gameplay and goal orientated gameplay.

4.4.1 Player Behaviour:

Object Identification, Pick up, Lift and Use Functionality:

Once the player could freely move using the Kinect control system, the first set of functionality to be implemented was identification, picking up and using objects. Ray casting was used to cast a ray in a positive z-axis direction from the centre of the screen (the player's crosshair). Hit information gathered by the ray cast would then be returned and used to output a string of the object's name to the player's head-up display (as shown in Figure 13, see below). Note, the ray has a limited length, which is set via a class member variable (m_itemReachRange).

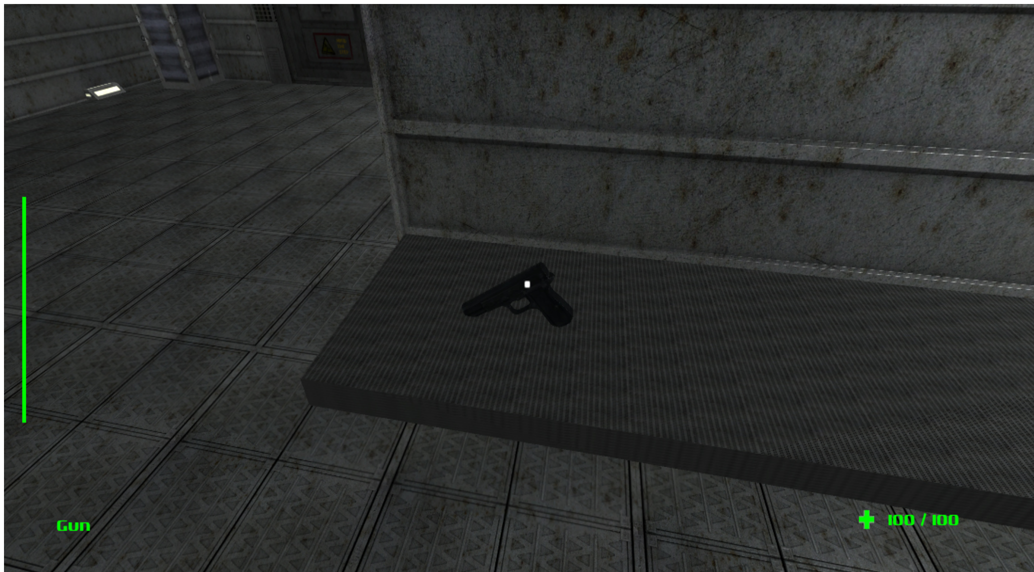


Figure 13: An image illustrating the object identification function. Note, the object's name printed to the HUD in the lower-left hand corner.

Within the game there are two types of objects, inventory items and non-inventory items, both of which can be picked up. Each item houses a different set of behaviours dependant on it's type and also name. For instance, an item which is of type “Inventory item” will be added to the player's inventory. Whereas, a item of type “Non-inventory item” can also be collected, but will have different behaviours dependant on it's name (i.e. both a battery pick-up, and health pick-up are non-inventory items, but invoke different behaviours). Once the player's crosshair is over the item, the player must then press the use/pick-up button in order to pick the item up. Note, the player's crosshair does not need to be *exactly* over the object, as a built in tolerance was implement; and as such the player can still pick up an object, if their crosshair is near enough to the object. This was used to aid inexperienced players, when aiming, to pick up an item with greater ease. Upon pick-up, non-inventory items invoke a behaviour (dependant on name) and are then destroyed (in order to use memory efficiently). Inventory items are added to the player's inventory (a dictionary collection) and are parented to the player, so that they can use them within the game (i.e. the gun and flash light will be positioned in player's hands, in order to use them). The pick-up button also constitutes as the player's use button, and can be used to open doors throughout the game. Additionally, as previously mentioned, players can use/pick-up using the hybrid control system (utilising the Kinect), by reaching out in front of their torso (see Section 4.3 Hybrid Control System - Xbox Control-Pad and Kinect Control System). Thus, a player can pick-up and item or open a door, simply by reaching out in front of their torso.

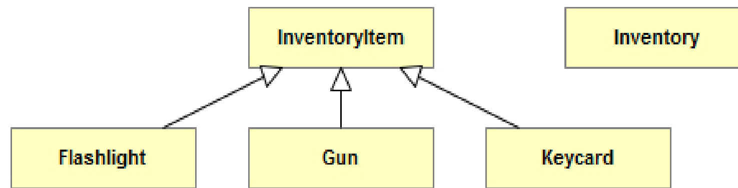


Figure 14: A diagram illustrating the inheritance structure of the inventory items used within the game.

All inventory items within the game inherit common properties from the InventoryItem class. Thus, simple items that do not have extra behaviour (for instance a secret/Easter egg item) can instantiate from this class. Additionally, creating new inventory items is an easy process due to the extendibility of the design, which is achieved via inheritance (see Figure 14, above).

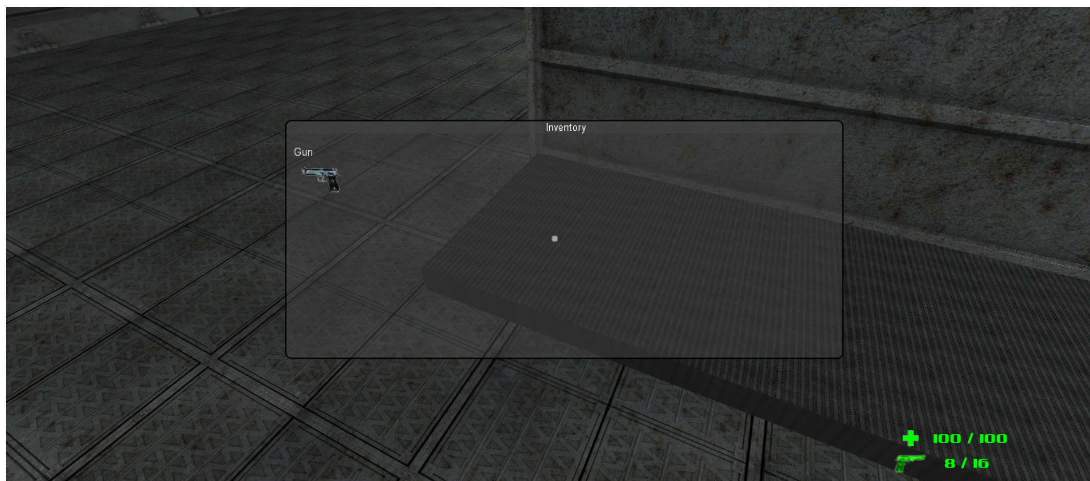


Figure 15: This image illustrates the gun inventory item being added to the player's inventory. The player can toggle the inventory GUI window on and off, by pressing a button on the control-pad.

The player's inventory consists of a dictionary collection, as each item is unique, and thus a player should not be able duplicate and collect more than one of the exact same inventory item. However, a player can collect a variant of a type of item (i.e. “Red Keycard” and “Green Keycard”).

Player's can also lift objects that are marked as “lift object” (as shown in Figure 16, below). Players lift objects within puzzle rooms to gain access to keycards (i.e. stack boxes to reach a platform), or to defend themselves against enemies (by using the object as a shield, blocking enemy fire). Items are lifted by simply setting their position to a specified length down the ray being cast (the ray which the player class casts). The object's velocity of it's rigid body component is set to 0, to ensure the universal gravity vector does not take effect (causing the object to fall back to the ground).

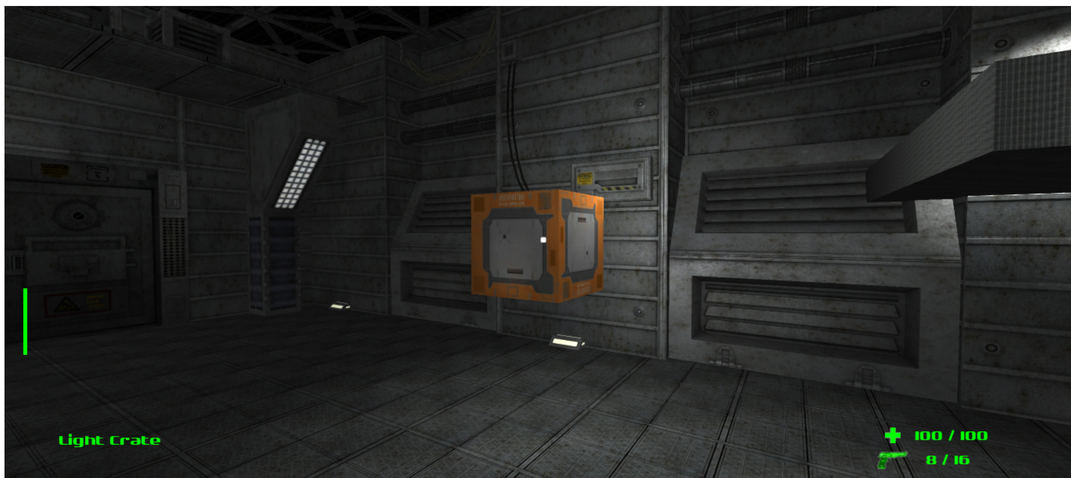


Figure 16: An image illustrating a “light crate” being lifted within the game.

Non-inventory items (i.e. pick-ups) consist of ammunition, health and batteries (as shown in Figure 17). Ammunition and batteries can only be picked up if the player has a gun or flash light in their inventory respectively.

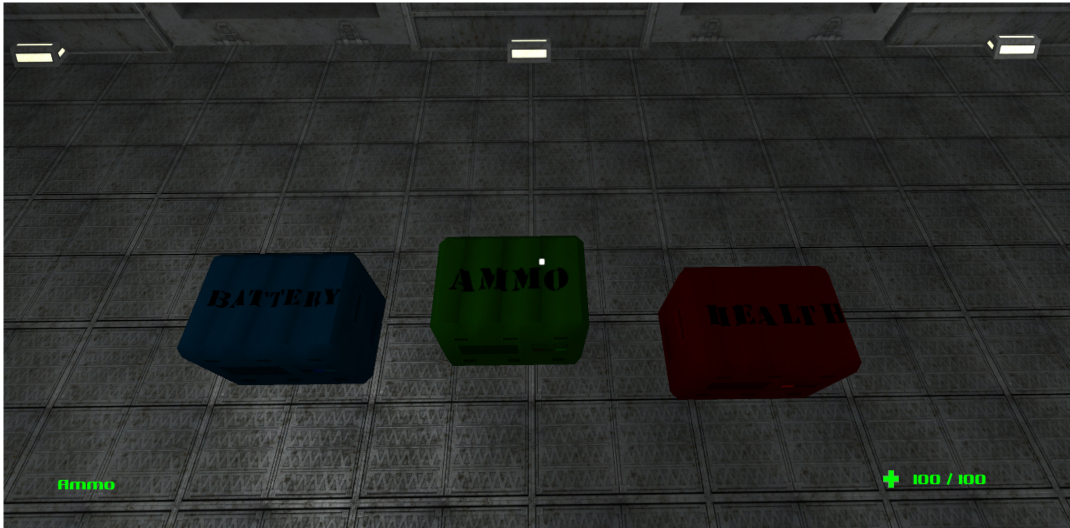


Figure 17: An image illustrating the 3 different types of pick-up within the game (battery, ammunition and health).

Player Damage and Death:

The player takes damage via enemy gunshot hits or meele attacks. The player takes different amounts of damage dependant on which area of their body they are hit. This is achieved through a number of collision boxes or “hit areas” positioned in a human body shape (see Figure 18, below). Thus, when a player is hit the method “takeDamage” is invoked, which consists of a switch statement, which in turn, decrements the player's health dependant on which body part was hit.

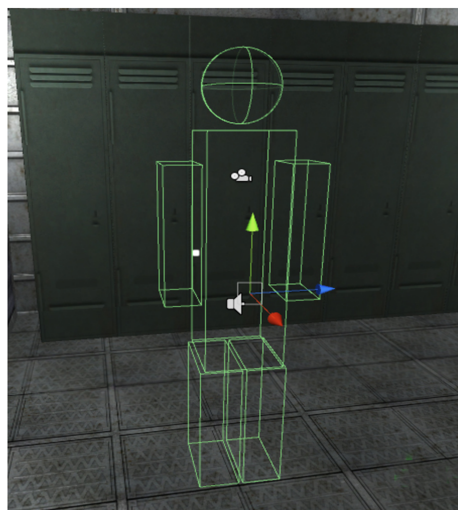


Figure 18: An image illustrating the player's different hit areas. Each hit area is a box collider, and detect hits from enemy attacks.

When a player dies they are transported back to the previous checkpoint (which is the last room completed. Note, rooms shall be discussed later in this thesis).

4.4.2 Flash Light:

The flash light can be collected at the start of the game, and serves as a tool to illuminate a room, when the lights within the level are turned off. The flash light consists of a spotlight, which can be toggled on and off throughout the game. This game mechanic increases atmospheric tension within the game, by affectively limiting the player's field of view (to the cone of the spotlight). The flash light itself has a power level, which drains, while the flash light is turned on. Batteries can be picked up to replenish the power level of the flash light. Once the flash light is collected, it's power level is made visible in the player's heads-up display.

Figure 19: An image illustrating the flash light turned on, within an area of the game where the lights are turned off. Note, the flash light power level displayed on the left-hand side of the screen.

4.4.3 Gun and Bullet Effects:

The player collects the gun at the start of the game, and uses it to defeat enemies and solve various puzzles within the game. The gun can be holstered and drawn by pressing a button on the game-pad (note, that objects cannot be lifted when the gun is drawn). The gun uses a ray cast system (the same methodology found within the player class for identifying and collecting items) to fire a ray (the bullet) and provides hit information

on any object the the ray collides with (i.e. any object the bullet hits). Thus, behavioural response from a bullet collision (the ray hitting an object) is dependant upon what ray hits. If the bullet (the ray cast) hits the surrounding environment (excluding objects that have been placed on a non-shootable layer, which include inventory items) a bullet hole and bullet spark are instantiated at the point the ray hit. Whereas, if a bullet hits an enemy, a blood splatter (from the exit wound) of the bullet is created. These, effects are known as "bullet effects" within the game, and reside in a separate class, so that the bullet effects can be used by both the player and enemies (that possess guns) throughout the game. If a player hits an enemy with a fired bullet within the game, the hit enemy will take damage (dependant on where they hit the enemy). Additionally, objects hit by a bullet will have a velocity force applied to them relative to the normal of the the point struck on the object (see Figure 20, below).

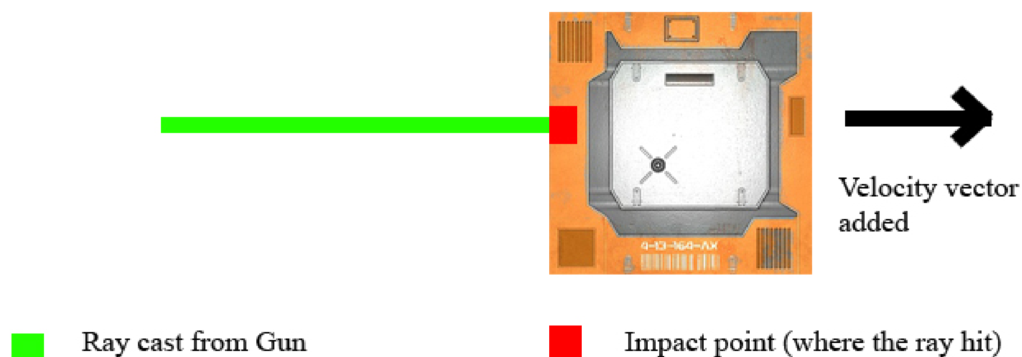
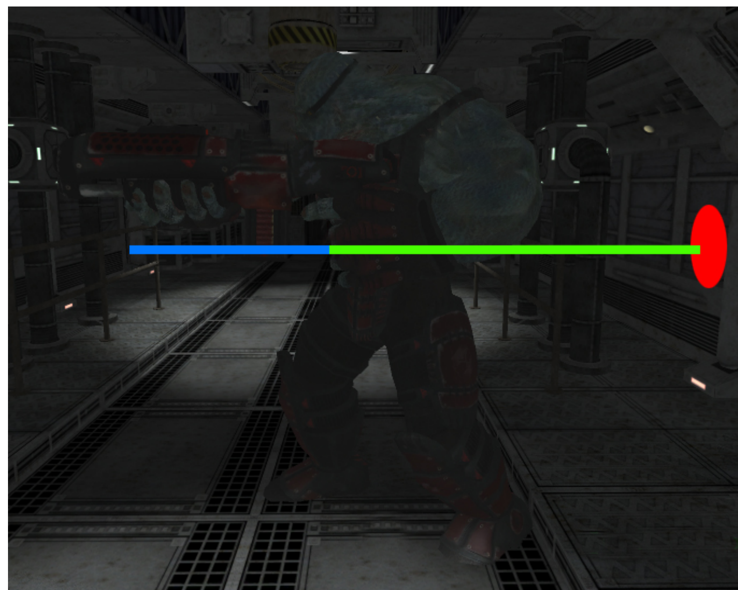


Figure 20: An image illustrating the addForce method, adding a velocity vector to the rigid-body of an object, from the bullet that struck it.

As previously mentioned, there are three different types of bullet effects, the bullet hole, bullet spark and blood splatter. The bullet hole is essentially a textured plane that uses a transparency shader to ensure the correct alpha levels are displayed (i.e. only the bullet hole is visible and not the rest of the plane). Upon the gun's ray hitting the surrounding environment, a bullet hole is instantiated, position and rotated dependant on the object or segment of the environment hit. A bullet spark object (which was created using Unity's in-built particle effects system) is then instantiated in the same manner as the bullet hole (and at same time).

Figure 21: An image illustrating a bullet striking the surrounding environment, and instantiating a bullet hole and sparks.

The blood splatter effect is created from the exit wound of a bullet passing through a enemy within the game. Upon the bullet (the ray) hitting the enemy, a new ray is created from the point of impact, which follows the same path as the initial ray cast from the gun to the enemy. The newly created ray passes through the enemy (and exits from the other side) eventually hitting the surrounding environment; at which point a blood splatter object is instantiated (at the point where the ray hit the environment).



Ray cast from Gun

Ray cast from where Gun ray hit (simulating the bullet passing through)

Blood splat instantiated when Ray passes through enemy and hits environment

Figure 22: An image illustrating a behaviour of a bullet striking an enemy.

All three types of bullet effects are destroyed after a pre-defined period of time, in order to efficiently use memory. Additionally, sounds effects of the gun firing (simply an audio played back when the gun is fired) and gun reloading (when the player presses the reload button) were also incorporated.

4.4.4 Enemy Behaviour:

Throughout the game the player will encounter enemies, which they will have to destroy in order to progress through the game. Enemies within the game are state-driven, dependant on several factors (i.e. health, ammunition etc.) and also share several common attributes (i.e. health, berserk and death functions). Thus, the a base class "Enemy" was created, acting a template class, for types of enemies (subclasses) to inherit from. Virtual methods were also interoperated, so that subclasses could override shared methods and implement their own alternatives. This ensures the enemy system remains flexible, while providing common functionality that a typical enemy would possess (see Appendix A: Class Diagram for Enemy and Orc classes).

Enemies (using the same methodology of how the player takes damage) take damage dependant on where they are hit (see Figure 23, below).

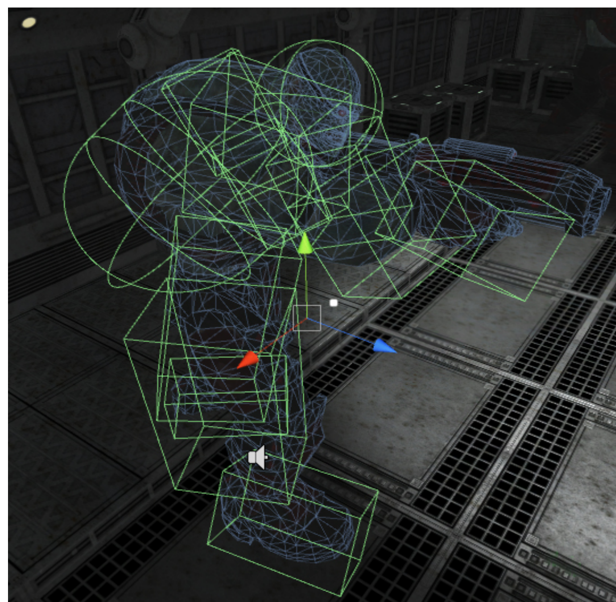


Figure 23: An image illustrating the orc's box collision hit areas. The hit areas determine where the player's bullet hit, and deals damage to the orc accordingly.

As previously mentioned enemies are state driven dependant on health or ammunition. Enemies will wait in an idle state until they detect a player (if a player is within a pre-defined range). Upon detecting a player, enemies will enter an attack state, in which they will attack the player. If the enemy reaches a pre-defined health level (below 30 percentage of it's maximum) it shall enter a berserk state, in which it charges the player to perform a meele attack. Once the enemies health reaches 0, it dies, and it's object is destroyed from the game (in order to use memory efficiently). Each state has it's own accompanying animation, using Unity's animation system and sound effect. Note, enemy animations were sourced from a third party (Dexsoft-Games, 2011).

Orc Enemy:

The orc enemy is currently the only enemy within the game (due to time constraints) but illustrates the enemy architecture's flexibility and extendible design. The "Orc" subclass extends the Enemy base class, utilising it's shared attributes whilst also using extended behaviours (implemented for uniquely for it's own use, such as firing it's gun). Using the enemy base class orc's also pass their own animations and sounds for each inherited method, enabling setting up enemy's behaviour easily accessible and quick.

The orc enemy uses a line casting system, when firing at a player (when it enters it's attack state), this is a special implementation of a ray casting system within Unity. A line casting system, takes two parameters the origin of the line to be cast, and it's destination. The line then returns any objects it intersects between it's pre-defined origin and destination points. Thus, the origin of a line cast emitted (when an orc fires it's gun) is from the orc's gun, to it's destination the player (once the player is within firing range, which is a pre-defined distance). Firing accuracy was added to ensure an orc does not hit the player every single time (thus realistically representing real world phenomena). The orc's firing accuracy works by taking the destination of the line cast (the player's torso) and adding a random number (defined between a random range) and adding to it's Cartesian co-ordinates (see Figure 24, below).

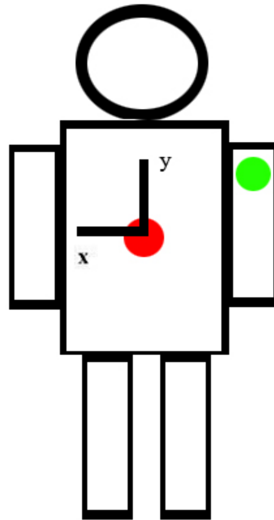


Figure 24: A diagram illustrating the orc's firing system, showing the original line cast destination, and the ultimate destination (with the random numbers added to the destination's Cartesian co-ordinates).

Friendly fire between orc's is disabled by choice (thus adding difficulty to the game, i.e. the player must use skill to destroy all enemies, and not rely on fortune). Additionally, orc's have a rate at which they can fire, once their ammunition in the current gun magazine is depleted they must reload their weapon. This is achieved by an orc crouching down and reloading it's gun with a new full magazine (the time taken to reload, is pre-defined by a local variable).

4.4.5 Door and Key System:

The door and key system is a common game mechanic, found within the many games across the gaming genres. Each door within the game requires a specific key-card in order to gain access (i.e. to open it). The key-card required is stored within a member variable within each door (i.e. the door's corresponding class, Door). The "m_keycardReq" variable itself is set via the name of the door. For instance, if the door's name is "Red Door" then the key-card required would be the "Red Keycard", or alternatively instead of using a colour system, "Section 1 Door" would require "Section 1 Keycard". The door checks if the player has access, by checking the player's inventory for it's required card. If the player's inventory does not contain the required

key-card, access to the door is denied (in which feedback is given, via a printed string in the player's HUD and a “access denied” sound effect). Additionally, a key-card's material (it's diffuse colour) within the game is set via it's name (i.e. a key-card with the name “Red Keycard” is shaded red). This method was simply implemented for quick deployment of a new key-card (without having to manual set it's diffuse colour each time) thus speeding up development of the game.

Figure 25: A image illustrating a key-card and door within the game.

4.4.6 Room Event System:

Room events are a common game mechanic which have been present among generations of developed games. Room events are game-driven events that feature within a pre-specified room, where a goal must be met, in order to progress within the game. Each room has an entrance door and an exit door. Once access has been granted to the entrance door, the room event begins. Once the goal of a room event is completed, a key-card will spawn (at a pre-defined point) which will enable the player to open the exit door of the room. Note, once room events are completed, the event itself is destroyed (there are two reasons for this, firstly as it is now redundant, and secondly in order for it not to be re-activated). Key-cards are spawned on tables (attached to the wall) this choice was made to ensure player's became familiar to where the key-card would spawned, once the goal was met. As opposed, to spending time aimlessly walking around the room, trying to find where it spawned. This method was suggested by a play tester, as part of qualitative research undertaken during usability testing, thus it was implemented (with the rest of the group members approving the idea).

There are two types of room events within the game, enemy rooms and puzzle rooms. Within an enemy room, a random number (using a pre-defined random range) of enemies are spawned. The goal of an enemy room event, is to destroy all the enemies in that room. Once the enemies are destroyed, the goal is met and the key-card is spawned. Within a puzzle room the key-card is spawned once the player opens the entrance door. Key-cards in puzzle rooms are spawned on platforms, in difficult to reach areas of the room. Thus, players must logically calculate how to reach the platform, where the key-card has spawned. Reaching the key-card typically involves lifting and stacking boxes, in order to jump on, to reach the platform. One room in the game also features a puzzle, where players must shoot barrels and crates off high platforms, in order to stack them to reach the key-card platform. Utilising a contrast (two differing types) of goal-orientated gameplay (i.e. shooting enemies and logically solving puzzles) provides the player with a more diverse and richer set of objectives to complete, within the game.

Figure 26: An image illustrating a puzzle room (left) and an enemy room (right).

4.4.7 Audio Events:

Audio events trigger sounds to be played within the game. These events are trigged, when a player collides with an audio event's box collider (a box which if the player enters, invokes the event). There are two types of audio event, a music event and a sound effect event. Music events are events which consist of a melodic (i.e. backing music) audio clip, which set the mood and atmosphere of the current scene within the game. Sound effect events trigger a sound effect to be played, which are typically short audio clips (i.e. an explosion or scream sound effect). Once these events are triggered,

the event's corresponding audio clip (the actual audio file) is sent to the player's "AudioReceiver" for playback. The AudioReceiver class simply plays an audio clip sent to it. Thus, using two different types of audio events, both music and sound effects can be mixed over each other, to create a richer audio experience for the player. Once an audio event has been played (i.e. the event has finished), it destroys itself (as it has performed it's task and is now obsolete). Note, that the audio event only sends information to the audio receiver and does not play the audio itself (the audio receiver performs this function). This approach was modelled after the model-view-controller model (the view being the trigger/box collider – i.e. the input, the controller being the audio event and the model being the AudioReceiver class).

4.4.8 Light Events:

Light events are trigged in the same manner as audio events (see Section 4.4.7 - Audio Events), through the use of box colliders. There are two types of light event, the first type is an on/off interval event, where the lights are temporarily disabled and then re-enabled, after a pre-specified length of time (a member variable defines the duration of the light event). The second type of light event is a "flicker" event, in which the lights flicker on and off (at a pre-defined flicker rate) for a specified length of time. The light events themselves do not control the lights directly, this is conducted by the LightsControl class. Thus, the light events send the LightsControl class information, that an event has been trigged and the LightsControl class invokes the events behaviours, by modifying the actual lights within the game. This approach (using the same methodology as the audio events) was modelled after the model-view-controller model (the controller being the light event, and the model being the LightsControl class) to ensure low coupling and high encapsulation.

4.4.9 Heads-up Display:

The player's heads-up display is another common game mechanic which was implemented as a part of the game. The HUD itself went through several iterations in response to play tester's feedback (as part of the usability tests). The HUD indicates necessary information the player must know throughout the game (such as current ammunition, health etc.) and no more. This is to ensure the player's viewpoint is not

overcrowded with any useless information and graphics, which would provide poor HCI and usability for the user.

Figure 27: An image illustrating the player's HUD, in which the three HUD elements (flash light, ammo and health statistics) have been highlighted.

4.4.10 Re-playability Factor:

A recent game evaluation study (Bernhaupt, Eckschlager, Tscheligi) states that "playability, namely that of repeated playability, especially after completion, can be seen as the *key* factor in a game's success". Thus, the "re-playability" factor within any game is always a desirable aspect for game developers to maximise. This is to ensure players continually replay their game, even after the player has completed it. Common mechanisms to achieve this, include recording and feeding back statistics to the player and secret elements (also known as, "Easter eggs", which are difficult to find when following the typical path of gameplay). Both of these mechanisms were implemented within the game. The game keeps a number of statistics throughout, such as: the number of times the player died, the time it took to complete the game and the number of secret items they collected. These statistics are presented to the player at the end of the game (see Figure 28, below). Thus, the player can replay the game to try and beat their personal best performances. Additionally, secret items have been hidden throughout the game, which can be collected by the player and added to their inventory (the secret items, within the game can be seen in Figure 29, below). These secret items again serve to ensure the player replays the game, in order to find all them, throughout the game. The "Easter egg" mechanism is a common re-playability mechanism found across all genre of games.

Figure 28: An image illustrating the end of game statistics screen, informing the player of their performance throughout the game.

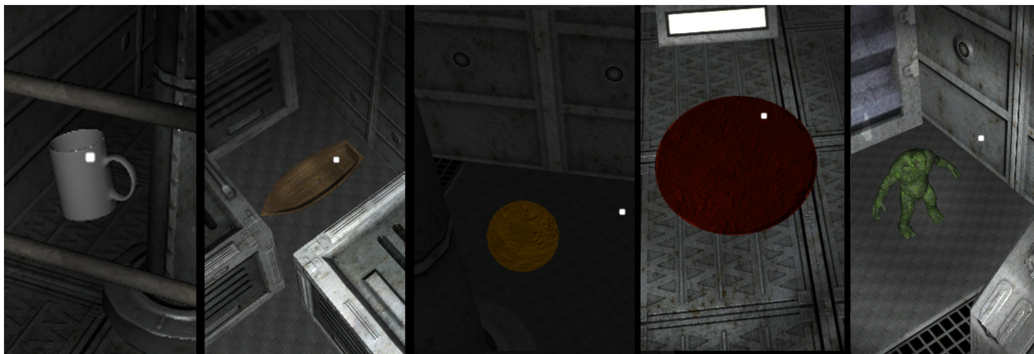


Figure 29: An image to illustrate the secret collectable items which are hidden throughout the game.

4.4.11 Game Settings and Main Menu

To complete the implementation an additional class (named GameSettings) was created, which enables easy modification of all the settings in the game. Furthermore, a main menu GUI was developed, which consists of an several buttons, two of which are a controls screen and an options menu. The control screen displays an image of the Xbox control-pad and informs the user of the control configuration (i.e. what actions, each button is mapped to). The options menu, consists of several GUI sliders and toggles, and serves as a GUI to the modify the variables of the GameSettings class (i.e. it enables the player to set/modify all the settings in the game – see Figure 30, below). Thus, the user can change the difficulty level of the game, by changing the difficulty of the enemies (i.e. their health) and also can change their own player settings (i.e. their maximum health, and starting ammo). Additionally, players can also change the Kinect settings, such as look sensitivity, enable camera roll and change hand orientation (i.e. the hand the player wishes to control the Kinect look functionality with).

Figure 30: An image illustrating the main menu of the game. The image shows the different sub-menus the player can open, and explore.

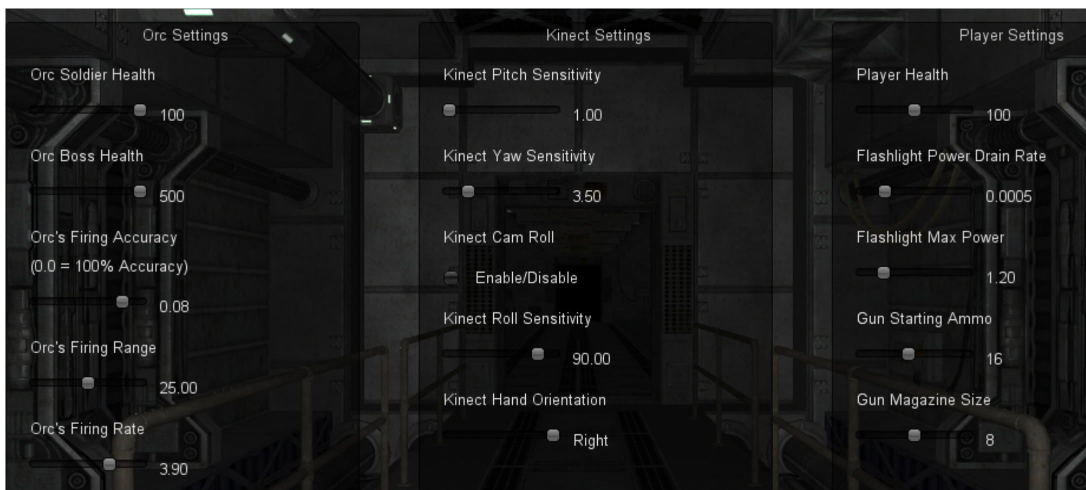


Figure 31: An image illustrating the options GUI (a sub-menu of the main menu). The player can use the GUI elements (sliders and a toggle) to modify the game's settings.

Once the game mechanics had been completed and tested, the surrounding environment (the level itself needed to be implemented). The level consists of a pre-built template package of corridors and rooms (Dex-soft Games, 2011). However, the entire level had to be arranged (from the pre-built corridors and rooms) and also lit, using Unity's lighting system. Audio, light and room events were then placed within the level, to create a deeper sense of immersion within the game (this was iteratively tested within the usability tests). Each room was then "set-dressed" with models appropriate to each room (i.e. puzzle rooms, monster rooms and rooms which narrate the storyline of the game).

All the code implemented adheres to NCCA coding standards and is also fully-documented using Doxygen. Furthermore, following industry standards, all implemented image files (with exception of those, sourced from Dex-Games Soft) are in TGA format, and all sound files are in WAV format.

The methodology when implementing the game was to ensure code was easy extendible and maintainable for any future development. Thus, designing several individual common game mechanics and collating them to create a game, has enabled the series of game mechanics to effectively become a framework. Thus, developers will be able to re-use the existing framework of code, to develop other games. This was achieved by creating code which embodies high encapsulation and low coupling, through use of design patterns (such as singleton classes) and inheritance.

5. Results of Usability Tests

As previously mentioned usability tests were conducted regularly throughout the development of the game, in the form of play testing. In addition to recording qualitative data (via player feedback and observation), quantitative data was also recorded. The quantitative data recorded, was in the form of game statistics (see, Figure 28). Upon the game being fully implemented, each play tester was asked to play both versions of the game (the Xbox 360 control-pad only version and hybrid control system version). Their statistics were then recorded, and are presented in graphical form (see Figures 32 and 33, below).

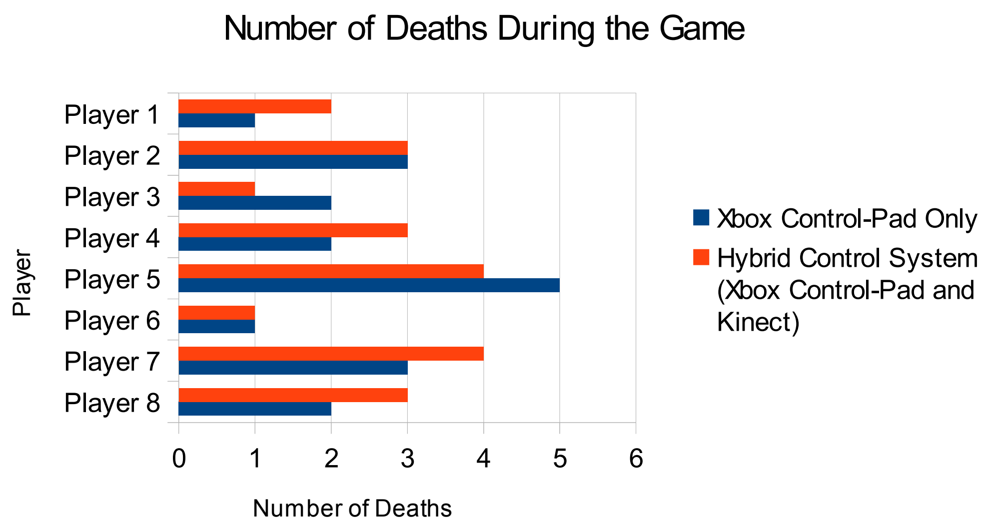


Figure 32: A graph illustrating the number of times a player died, within both versions of the game.

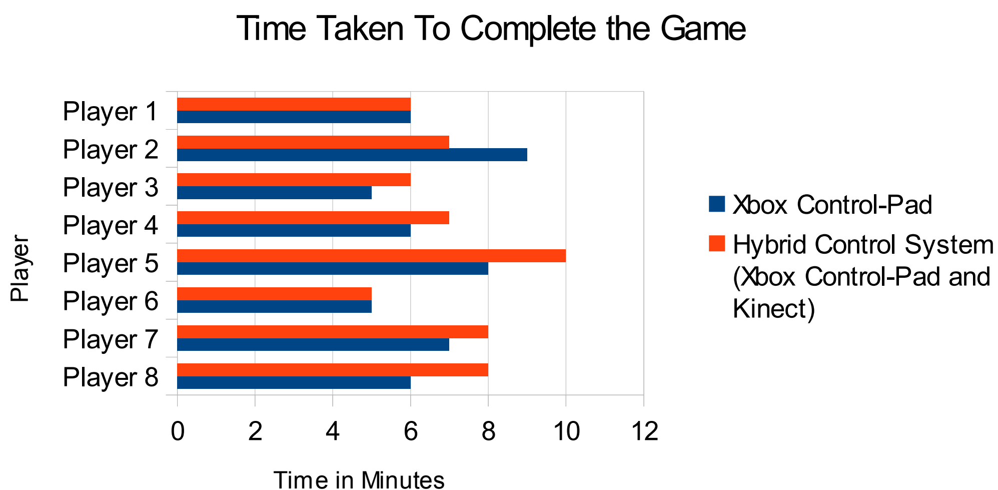


Figure 33: A graph illustrating the time taken by a player to complete the game (times for both versions of the game, have been recorded).

The reasoning behind both types of data (qualitative and quantitative) being recorded, was to gather insight into how the differing control systems compared to each other. In terms of qualitative data, experienced players were typically sceptical at first, of using the hybrid control system. However, once they started to become accustomed to the control system, they expressed they found it both enjoyable and intuitive. The quantitative data states that players performed better with the Xbox control-pad only. However, players (especially experienced players) expressed that they generally felt they performed better, due to their previous familiarity with the Xbox control-pad itself. Inexperienced players expressed their enjoyment using the hybrid control system equally, in relation to their more experienced counterparts. Players within the group also expressed that they would now prefer to use the hybrid control system over the traditional Xbox game-pad, due to the extra variety of controls and enjoyment the control system provides.

6. Conclusion:

The projects' objectives, to critically analyse and compare the two control systems through usability testing via a developed game, was successfully met. The developed game proved an ideal testing bed (and reflected the functionality found within that of a typical mature game), for usability tests to be conducted, within a small group of play testers. Through, the results of usability tests, insights were drawn, into how the two control systems compared, within a developed game. The usability results, illustrated that both inexperienced and experienced players found the hybrid control system both intuitive and enjoyable to use. However, the key limitation to the control system is that players have become widely accustomed to using only the traditional Xbox game-pad. Therefore, whilst the hybrid control system proved enjoyable amongst players, the biggest challenge it faces, is to persuade players to adopt it over only using the traditional Xbox game-pad. Although, the usability tests also reflected, that players would adopt the new system, in favour of the only using the traditional Xbox game-pad, due to the added depth and enjoyability, the hybrid control system brought. Thus, it can be concluded that, the hybrid control system does have a reputable chance of becoming an established control system on the Xbox 360 platform.

A key limitation of the study, was the limited number of participants within the usability testing group. Thus, it can be drawn, that provided with a larger number of participants, the results would have proved more accurate, and provided a greater insight into the comparison of the control systems. Furthermore, a larger number of participants would have provided further clarity, with regards to whether the hybrid control system, would have been adopted by the mature gaming community (either in conjunction, or in favour, to it's traditional counterpart, of only using Xbox 360 control-pad).

In terms of expanding the project, enabling multiple players to use the hybrid control system simultaneously, would be the next logical step. Players would be able to play with each other locally or via a network. The network itself would be Microsoft's Xbox Live (Microsoft, 2002) service, which would enable players to play the game (using the hybrid control system) over the network. The game itself could be deployed via Microsoft's Live Arcade (Microsoft, 2004) enabling player's to download it to their Xbox 360 games console. Enabling the game to be readily downloaded by the entire

Xbox gaming community, would provide a large test sample of players, to test the hybrid control system (and would ultimately give a true reflection, by the gaming community, of how it compares, to only using the traditional Xbox 360 game-pad).

The multi-player functionality itself would be implemented using the Unity engine's in-built network framework. Methodology, such as state synchronization and remote procedure calls would be required, to be implemented between the sever and client machines. The server typically would be an allocated player's Xbox 360 game console, where client's (other player's Xbox 360 consoles) would connect, via the Xbox Live service.

7. References:

Microsoft, 2010., Kinect Operation Manual. Available from: http://support.xbox.com/en-gb/pages/xbox-360/get-started/manuals-specs.aspx [Accessed 28 July 2011].

Rubin, J. and Chisnell, D., 2008. Handbook of Usability Testing: How to Plan, Design and Conduct Effective Tests. 2nd Edition. John Wiley and Sons.

Bernhaupt, R. Eckschlager, M. and Tscheligi, M. 2007. Methods for Evaluating Games: How to Measure Usability and User Experience in Games. Available from: http://dl.acm.org/citation.cfm?id=1255142 [Accessed 4 August 2011].

Muich, N., 2006. Xbox 360 Controller. Flicker. Available from: http://www.flickr.com/photos/not-so-much/3696020923/ [Accessed 18 July 2011].

Henry, B., 2011. Kinect Adventures Rally Ball. Sitsam. Available from: http://www.sitsam.com/wordpress/wp-content/uploads/2010/11/Kinect-Rally-Ball.jpg [Accessed 18 July 2011].

Brutal Gamer, 2010. Player using the Kinect. Brutal Gamer. Available from: http://brutalgamer.com/wp-content/uploads/2010/11/player-considerations.gif [Accessed 21 July 2011].

IGN, 2011. Harry Potter and the Deathly Hallows game (Kinect Side-Mission). Available from: http://360.mmgn.com/Lib/Images/News/Normal/Kinect-compatibility-for-latest-Harry-Potter-adventure-1049385.jpg [Accessed 23 July 2011].

VaroLogic Blog, 2008. Player holding Wii Remote and Nunchuk. Available from: http://www.varologic.com/blog/Images/Wii-remote-and-nunchuk.jpg [Accessed 24 July 2011].

Kotaku, 2011. Player holding PlayStation Move and navigation controller. Available from: http://cache.gawkerassets.com/assets/images/9/2010/03/move3_01.jpg [Accessed 24 July 2011].

Syfan Media, 2011. Red Steel Wii game box art. Available from: http://www.syfanmedia.com/images/boxshoredsteelrge.jpg [Accessed 26 July 2011].

IGN, 2011. Killzone 3 PlayStation 3 box art. Available from: http://ps3media.ign.com/ps3/image/article/109/1099404/killzone-3-20100616055042869.jpg [Accessed 28 July 2011].

Tribal Labs, 2011. OpenNI skeletal tracking points diagram. Available from: http://www.triballabs.net/wp-content/uploads/2011/06/Kinect-skeleton-tracking.png [Accessed 29 July 2011].

Microsoft Studios, 2010. Kinect Adventures. [video game]. Microsoft Studios.

Sega, 2011. Virtua Tennis 4. [video game]. Sega.

EA Bright Light, 2010. Harry Potter and the Deathly Hallows - Part 1. [video game]. Electronic Arts.

Rockstar North, 2008. Grand Theft Auto IV. [video game]. Rockstar Games.

Ubisoft Paris, 2006. Red Steel. [video game]. Ubisoft.

Guerrilla Games, 2011. Killzone 3. [video game]. Sony Computer Entertainment.

All models, animations and art work, used within the game are sourced from:
Dex-Soft Games. 2011. Models, animations and art work [game art assets]. Dex-Soft Games. Available from: http://www.dexsoft-games.com/ [Accessed 29 July 2011].

All sounds used within the game are sourced from:
Free Sound. 2011. Sound effects and music [sound assets]. Free Sound. Available from: http://www.freesound.org/ [Accessed 1 August 2011].

Unity. 2011. Unity Game Engine [computer programme]. Unity Technologies. Available from: http://unity3d.com/ [Accessed 18 June 2011].

OpenNI. 2011. OpenNI Framework and NITE [framework and middleware]. OpenNI. Available from: http://www.openni.org/ [Accessed 20 June 2011].

8. Bibliography:

Dix, A., Finlay, J., Abword, G. and Beale, R., 2003. Human Computer Interaction. 3rd Edition. Prentice Hall.

Rubin, J. Chisnell, D., 2008. Handbook of Usability Testing: How to Plan, Design and Conduct Effective Tests. 2nd Edition. John Wiley and Sons.

Gold, R., Skelly, T. and Theil, D. 1994. What HCI Designers Can Learn From Video Game Designers. CHI '94 Conference Companion on Human Factors in Computer Systems. Available from: http://dl.acm.org/citation.cfm?id=260220&bnc=1 [Accessed 2 August 2011].

Barr, P. Noble, J. and Biddle, R. 2006. Video Game Values: Human-Computer Interaction and Games, Interacting with Computers. Volume 19 (2), 180 – 195. Available from: http://www.sciencedirect.com/science/article/pii/S0953543806001159 [Accessed 5 August 2011]

Jorgenson, A. 2004. Marrying HCI/Usability and Computer Games: A Preliminary Look. Available from: http://dl.acm.org/citation.cfm?id=1028078 [Accessed 4 August 2011].

APPENDICES

Appendix A - Class Diagram: