|  |
| --- |
| Staffordshire University |
| Advanced Windows Games Programming CE00391-6 |
| Development of Windows game software. |
|  |
| **By Alex J Davison (DV003874), Daniel Chapman (CU002003) & Simon Layton (LV002907)** |
| **1/6/2012**  **Word Count: 4663** |

|  |
| --- |
|  |

Contents

[Introduction 3](#_Toc312445128)

[Component Design 3](#_Toc312445129)

[Input 3](#_Toc312445130)

[Physics 5](#_Toc312445131)

[Graphics 10](#_Toc312445132)

[Testing Plan 13](#_Toc312445133)

[Unit Testing 13](#_Toc312445134)

[Integration Testing 14](#_Toc312445135)

[Regression Testing 16](#_Toc312445136)

[Critical Evaluation 17](#_Toc312445137)

[Project Management 18](#_Toc312445138)

[Overview of the Engine 20](#_Toc312445139)

[Racing Game 20](#_Toc312445140)

[Plat-former Game 20](#_Toc312445141)

[Goop Game 21](#_Toc312445142)

[Bibliography 23](#_Toc312445143)

[Appendices 24](#_Toc312445144)

# Introduction

This report covers the design of graphics, input and physics components for a game engine. The three components that were created were a graphics, input and physics component. This report will also cover how these components were tested, a critical evaluation of the game engine in its entirety, how the work was divided among the three programmers, how the project was managed and finally, how the game engine was used to develop three game prototypes.

# Component Design

include diagrams (E.g. UML diagrams) of your components and the design choices/patterns utilised

In this section of the report we will cover how the components were designed, decisions made and/or design patterns used in the component design. An overview of the integration of these components into a single game engine is also presented.

## Input

The input component was broken into six main classes. One called the InputManager which is responsible for informing the game of player events. A player event object which is a light weight object for packaging data for the game to use. A player object that contains all the data regarding the player’s control, moves and uniquely identifies a player. A player move object which is a light weight object used to match a move with a control. A three remaining classes are for the three different types of input that are supported, the keyboard, mouse and Microsoft’s Xbox 360.

The reason for the three input classes was due to the different values that are created from the input devices. The keyboard only handles digital value for a key press i.e. the key is either pressed or not. The mouse handles digital button presses like the keyboard but also gives the position of the mouse curse on the screen in the X and Y axis as well as a value for the mouse wheel. The gamepad however creates digital and analog values for the two joysticks on the controller.

Due to the different values that the different devices feedback to the computer, meant that three different classes where better suited to handling the input. This also allowed another principle of handling the input, which was to only create objects that were only needed to monitor the input devices used i.e. if the keyboard is being used then monitor otherwise do not check the keyboard for change. Input objects then fire event when a change had been detected. The input manager would listen for event from the devices that had been created. The input manager then creates a generic object for the game to handle. The process is as follows:

1. Call the update from the input manager.
2. Update all the instance of the controllers that have been initialised.
3. Check to see if controller has change since last time they were updated.
4. Check what needs to be checked for the player i.e. Does the ‘A’ button need to be if the player has allocated a move to that button, if so check it.
5. Check what has changed and how i.e. ‘A’ button has been pressed and ‘B’ has been released
6. Inform the inputmanager that the controller has changed state.
7. Find out which player uses the controller and what game move is assigned to the element that has changed.
8. Create a light weight object for the game to handle with only relevant information i.e. player one has started to kick or player two has moved forward fast.

Figure 1 - Example of player event from testing show the light weight object in text form that is sent to the game from the input manager.

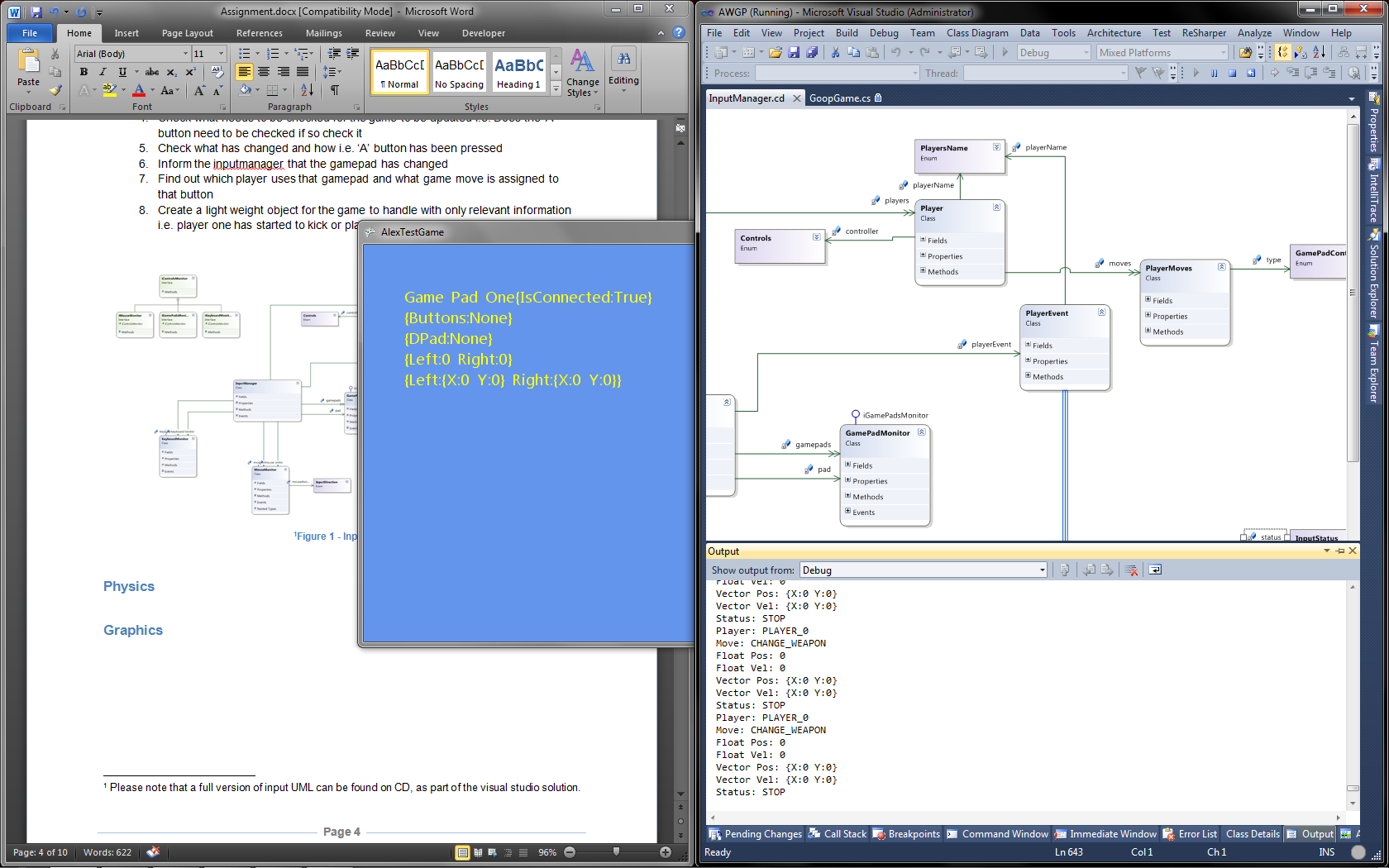
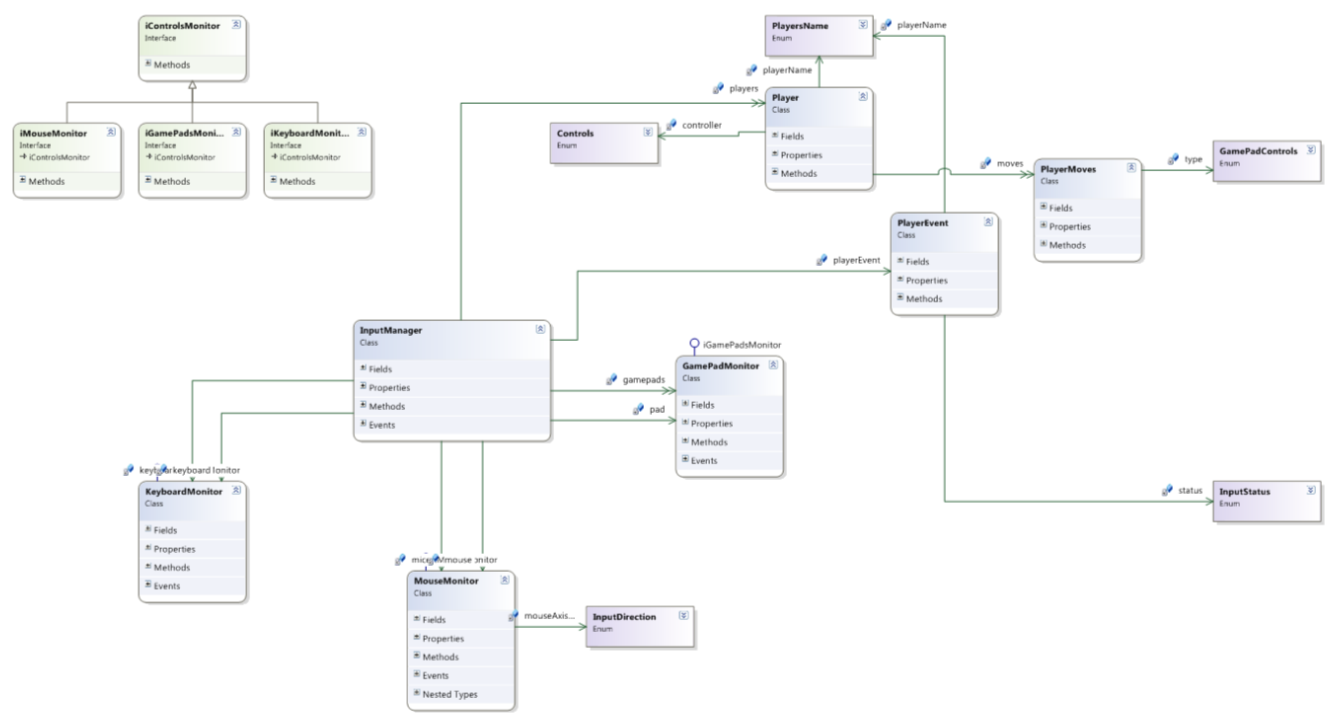


Figure 1 - Example of player event from testing

* Player – identifies which player created the event.
* Move – tells the game what the player wants to do.
* Float Pos – is used for the analog value such as [[1]](#footnote-2)acceleration, which can be passed to the physics of the game, used by the gamepad and mouse for the trigger and mouse wheel
* Float Vec – is used to tell the game how fast the controller element was pressed since the last update. This can be used to apply harder acceleration, this is used by the gamepad and mouse for the trigger and mouse wheel.
* Vector2 Pos – are like Float Pos but with two axis, this can be used for camera movement etc. or for the mouse where the mouse is on the screen. These are use for the mouse and joystick elements.
* Vector2 Vel – gives the velocity of how fast the joystick or mouse was moved since its last update.
* Status – tells the game whether the movement of player event is a new or old event. This can be used to remove a force given to an object or add a force to a game object.



[[2]](#footnote-3)Figure 2 - Input UML

## Physics

The Physics component of the engine is divided into subsystems to deal with forces, collisions, movement and particle effects. The system provides the following features:

* Rigid Body physics simulation
* Event driven collision detection and response
  + Including simulation of elasticity in collisions
  + Using bounding rectangles and circles.
* Force system
  + Including rotation forces

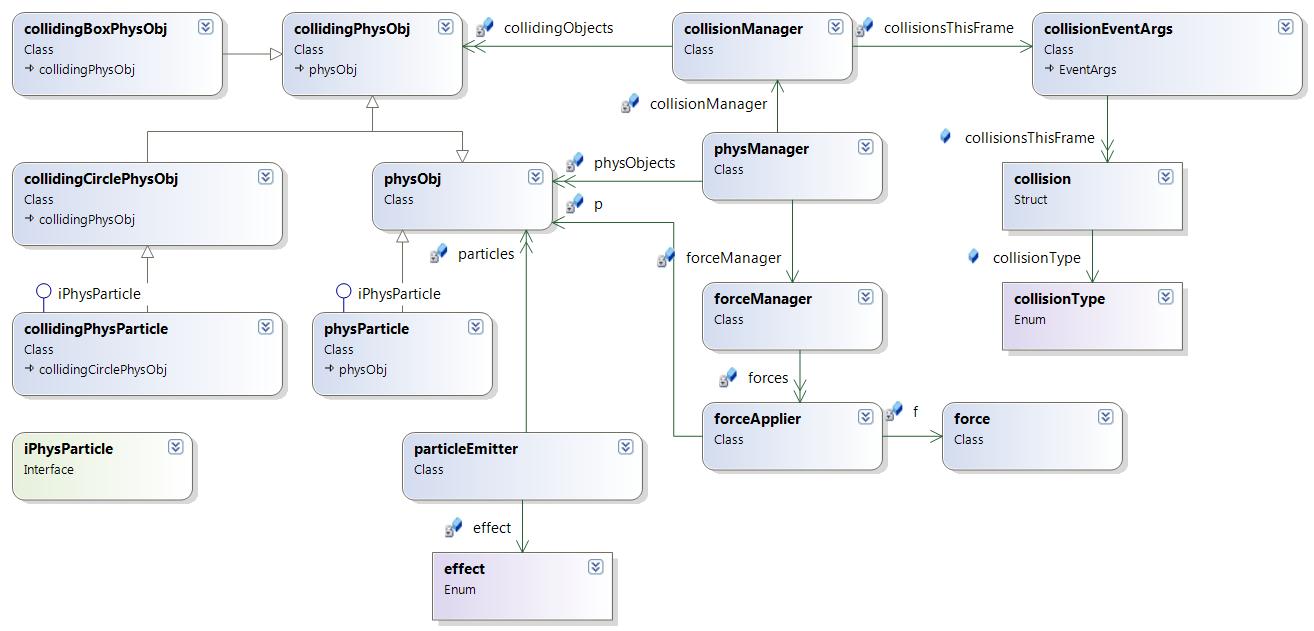


Figure 3 - Physics Component UML Class Diagram

The Main class of the system is the physManager. A physManager object is held by the game and provides access to the physics subsystems.

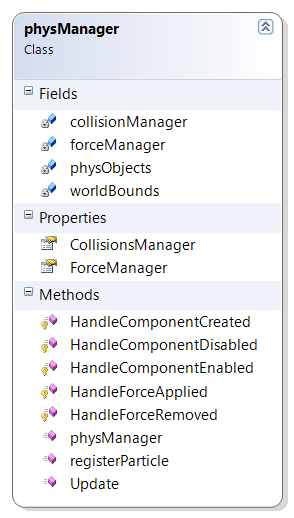


Figure - physManager UML diagram

Contained in the manager are the references to the collision and force managers as well as public methods to Update the world and register new physics objects with the system.

It also keeps a list of all of the physics objects in the scene and updates their positions every frame as part of the update loop. The physManager.Update() method should be called every frame by the game code. The method then updates all of the physics subsystems and returns control back to the game code until the next frame.

The Update method is as follows:

public void Update(ref GameTime gameTime)

{

forceManager.applyForces(gameTime);

foreach (physObj ob in physObjects)

{

ob.update(ref gameTime);

}

collisionManager.checkCollisions();

}

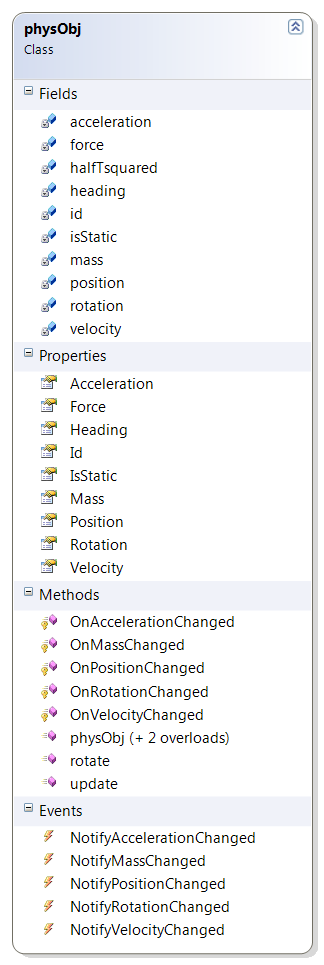


Figure - physObj

The method begins by calling applyForces in the forceManager, which applies and updates the forces acting on the physics objects in the scene. It then iterates through the list of physics objects and calls their update methods, so that they update their individual positions. Finally it tells the collision manager to check for collisions between objects in the scene.

The physObj class represents an object in the scene. The current acceleration, mass, position, rotation and velocity of the object are stored as well as an ‘id’ String so the game can tell the objects apart. The Vector2 variable force is updated by the force manager when it resolves all of the forces acting on the object at the start of the physics update. The heading attribute stores a normalized vector representing the direction that the object is facing. The rotation property is used by the graphics system to orientate the sprite when rendering.

There are also several events that fire when attributes of the object change. The graphics system is subscribed to these to update the position of the sprite that is drawn to the screen.

The Update method uses the following algorithm to compute the motion of an object:

halfTsquared = deltaT \* deltaT \* 0.5f;

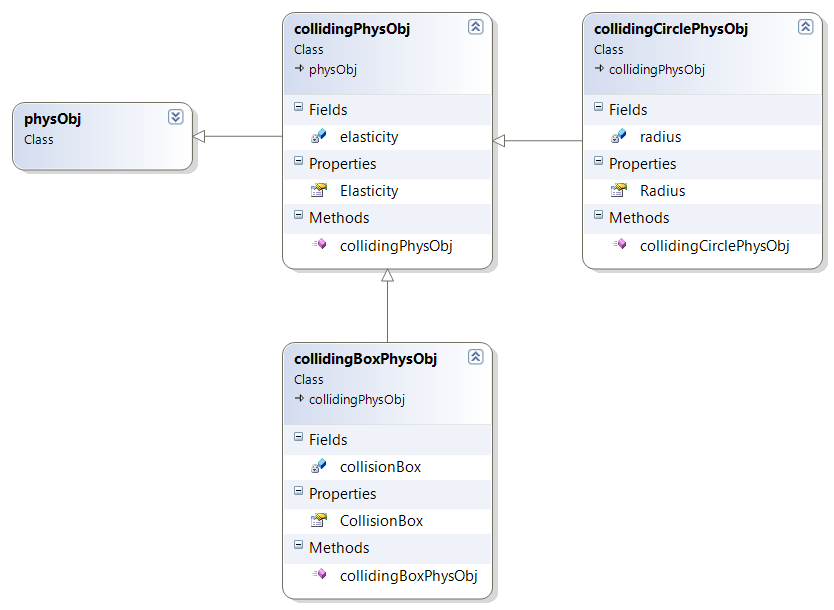
Acceleration = Force / Mass;

Position += Velocity \* deltaT + Acceleration \* halfTsquared;

Velocity += Acceleration \* deltaT;

This is based on the Newtonian motion equation and Newtons second law of motion,. The value of is calculated each frame due to possible variations in the frame rate during execution. In a fixed frame rate environment, the value could be calculated once at run time as it would be the same every frame.

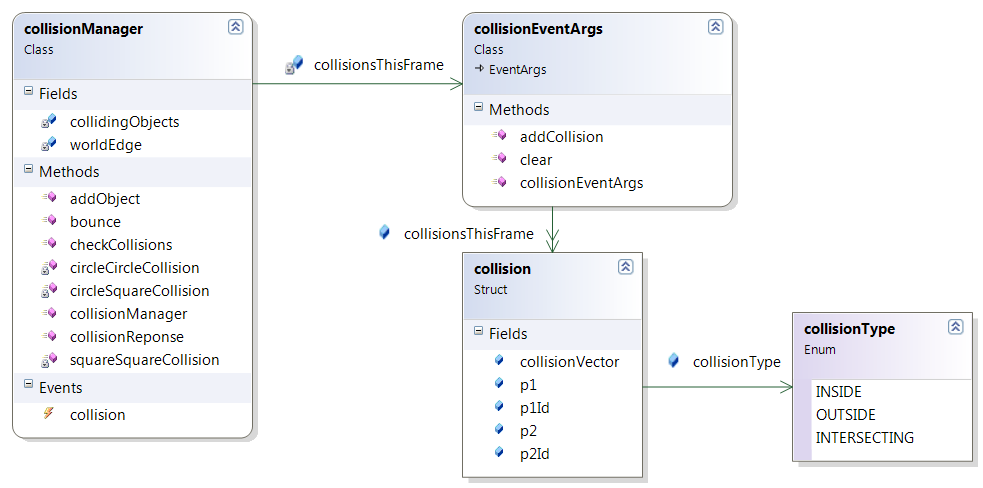
Angular motion is not accurately simulated, but a heading vector and rotation angle are provided primarily for the graphics system to orientate the sprites. The heading vector can also be used by the force system to apply a force in the direction the object is heading, (such as acceleration in a vehicle.



Figure

For the purposes of collision detection there is a subclass of PhysObj called collidingPhysObj. It serves 2 purposes; the addition of an elasticity value for use in collisions and to act as the base of the colliding object hierarchy. The class is not used to represent a particle, instead two more subclasses of collidingPhysObj are provided to provide a physics object with a bounding rectangle or circle, called collidingBoxPhysObj and collidingCirclePhysObj respectively.

The bounding circle in CollidingCirclePhysObj is represented by a radius, with the centre of the circle being the location of the physObj. The bounding rectangle in a collidingBoxObj is represented by a Vector2 (width & height), with the centre being in the location of the physObj.



Figure

The collision manager holds a list of all of objects in the scene descended from collidingPhysObj. When checkCollisions() is called, the collisionManager iterates through the list of objects and creates a collision object for each pair of objects. A collision can be in three states: inside, where one bounding shape is totally within the other; outside, where the bounding shapes are not colliding at all and intersecting, where the edges of the bounding shapes are intersecting at one or more points.

The collisions are all added to a list in collisionEventArgs and sent with the collision event which is fired once all of the collision checks have been made. When the collision manager is constructed 4 bounding boxes just off the edges of the screen are created and added to the list of colliding objects automatically, so world edge collisions can be handled. The checkCollisions() method determines the types of bounding shapes in the collision and calls static methods in the collision manager to create the collision object. There are separate methods for rectangle – rectangle collisions, rectangle – circle collisions and circle – circle collisions. The resulting collision object contains references to the colliding physics objects, a vector between the positions of the objects and the type of collision.for the different

The physics components subscribe to the collision event and pass only the collisions involving that object to the OnCollision() method.

## Graphics

Graphics are an important part of any video game, and as such careful consideration was necessary when deciding what features were required or desirable for a flexible game engine. Given that it had already been decided that the physics component would be a two-dimensional due to the difficulty of accurately simulating complex shapes and volumes in three dimensions, this meant that 3D graphics would be a poor fit for the engine’s functionality. It was decided that implementing a simple deferred rendering engine would provide a flexible and powerful 2D graphics component, which would render many sprite and light instances simultaneously from which visually interesting scenes could be constructed.

Deferred rendering refers to a number of techniques which have gained mainstream popularity in modern console and PC games in recent years, and was arguably popularised for use in video games following the implementation used in S.T.A.L.K.E.R. was documented in GPU Gems 2 (Shishkovtsov, 2005). Although implementation details differ, a deferred renderer generally renders the scene in two passes; firstly geometry information such as surface position, normal and un-lit colour is rendered to intermediate buffers known as collectively as a Geometry Buffer (or G-Buffer). The second pass applies lighting calculations to the scene by rendering each light as a screen-space post-process effect, using the information stored in the G-Buffer along with per-light information to apply standard 3D graphics lighting algorithms, and accumulating the results of these lighting calculations into a final image buffer. There are benefits and drawbacks to this approach, as compared to a more traditional forward renderer, however for our purposes the main benefit is the ability to efficiently render large numbers of lights independent of scene complexity.



Figure 8 – Table showing the chosen G-Buffer Format

[Explain my implementation of this concept & justify choices made]

[Present overview of system design, including use of design patterns & diagrams to show relationships]

## Component Integration

As the individual game components are developed separately, when using these components as part of a game it became necessary to manage their integration into a larger engine or framework.

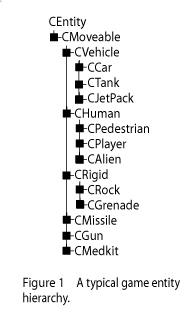


Figure 9 – Example of a typical game entity hierarchy (West, 2007)

Traditionally, objects in the game world are represented using some form of ‘entity’ class, with additional functionality being added when through the use of an inheritance hierarchy. At the base of this inheritance tree often lays some form of base entity class, with the leaves of this tree each representing a distinct type of game object. Whilst this architecture has been used in many successful shipped titles, it does have some notable drawbacks. With these deep inheritance hierarchies, there can be a tendency for functionality to pool either at the root or at the leaves of the tree; excess functionality in the root entity class can result in functionality being unnecessarily presented to all derived objects, causing overhead in those which do not use the it, whereas excess functionality in the leaf classes reduces code re-use and can result in monolithic, ‘special-case’ entities such as player entities. Additionally, due to the unpredictable, iterative nature of game development, it is not uncommon for new game entities to be designed which require functionality which is encapsulated in separate branches of what had previously appeared to be a sensibly designed inheritance hierarchy. This can require potentially messy solutions such as multiple inheritance, replicating functionality in the leaf node or even rewriting much the inheritance hierarchy in order to accommodate the new object type.

An alternative approach is to add functionality to game entities through the use of aggregation, rather than inheritance, sometimes referred to as a component based entity system. Please note that in this context, a component refers to a unit of functionality or set of properties which can be added to a game entity, rather than a full game engine subsystem such as the graphics, input and physics components implemented in this project. For clarity, the term ‘entity component’ will be favoured in this report to distinguish between these usages, though the terms are used interchangeably in the developed code, as other contextual information often makes the usage implicit. With this architecture, entities become little more than a shallow container for these entity components, with some implementations eschewing an entity class altogether in favour of logically-coupled but independent entity components. Each distinct game entity type is therefore constructed of varying combinations of these entity components, vastly simplifying the code for each entity. It would be possible to use a scripting language at this stage to define each entity type, offering vastly improved iteration time and flexibility to developers using the engine, although due to time pressures, this was outside the scope of this project.

Mick West (Evolve Your Heirachy, 2007) discusses his experience with using a similar architecture with the Tony Hawk series of games at Neversoft. He concludes that although there are some practical implementation difficulties, especially when transferring from an existing entity hierarchy, that the architecture increases the flexibility, re-usablity and robustness of the code.

Our game engine handles the integration of components through the use of a similar entity component system. Game entities can be created directly as entity components, or as shallow classes which encapsulate the creation of linked entity components, and any game-specific code which acts upon them. When each entity component is created, the component that processes or handles it is informed via the use of a static event, at which point the component can store this in whatever data structure is most appropriate for this entity component type, and continue to handle it independently of whatever entity type it is a part of. In this way, the integration of components can be kept as much as possible in game code, rather than engine-side code. This allows the components to process their logic completely independently, whilst still presenting flexibility to the game to handle their interactions. Our implementation heavily utilises C#’s event and properties functionalities to handle this integration, with entity components exposing sets of events and properties which can be handled as desired by game code. Events from entity component can be registered with either handlers from other entity components, in order to handle inter-entity-component integration, or with custom handlers in order to handle game specific logic.

# Testing Plan

describe and document the testing process undertaken to ensure quality of coding, including details of any unit, integration and/or regression testing undertaken.

In this section of the report we cover the testing that was carried out and how the test plan was conceived.

## Unit Testing

Unit testing was split so that each person was responsible for the testing of their own component/unit. Examples of this test are as follows.

### Input

Figure 3 - XML Testing for input shows and XML that was used to test the input. This was the data used to setup the input manager. As you can see testing of all the buttons for the gamepad was covered. The keyboard and mouse were tested in the same manner.

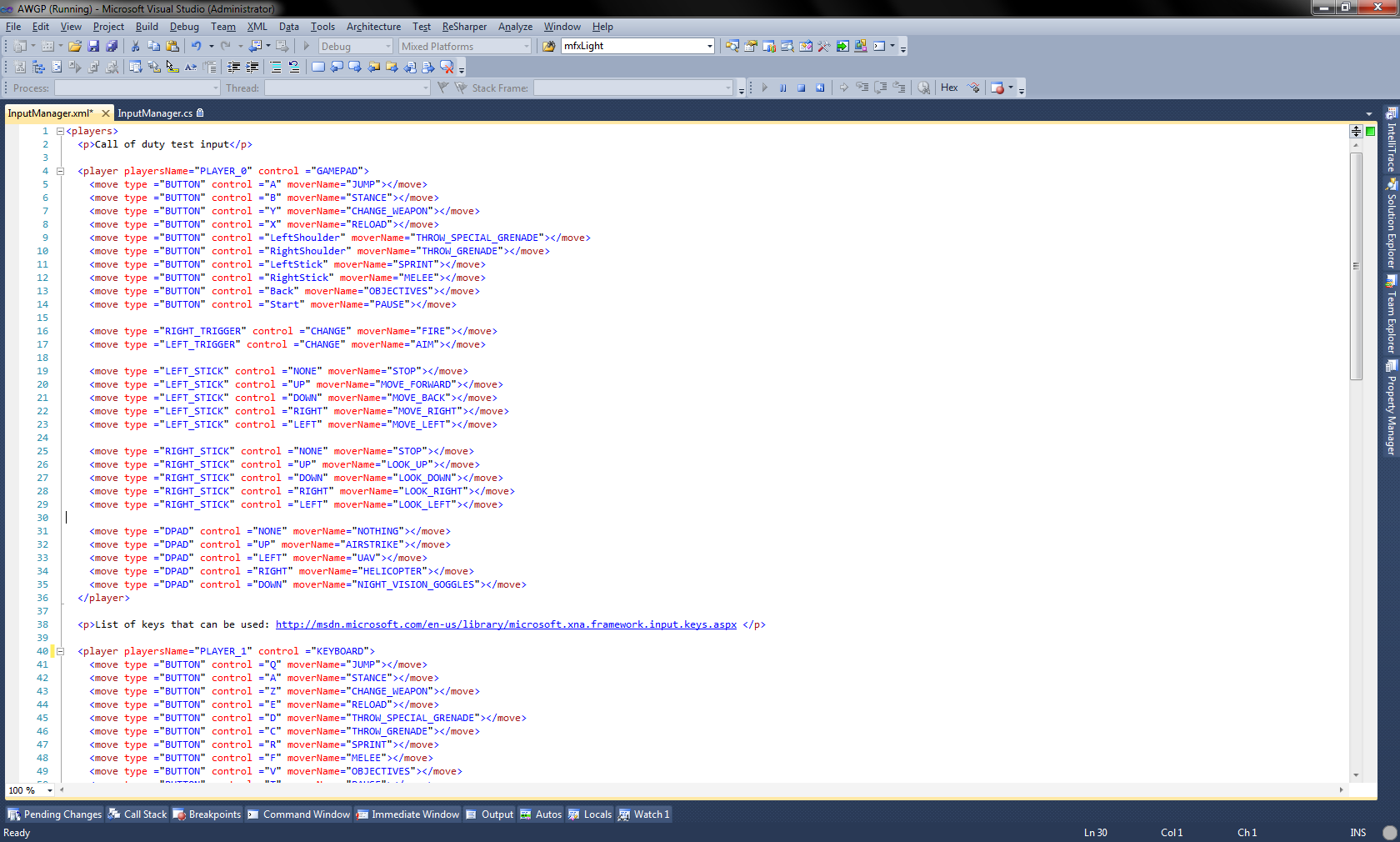


Figure 10 - XML Testing for input

A test was developed to display the really time information coming directly from the input. When the buttons, joystick and or another input was pressed this information would display with in the test game as shown by Figure 4 - Input test game. This meant that a really string version was displayed so that it would be checked against the player events.

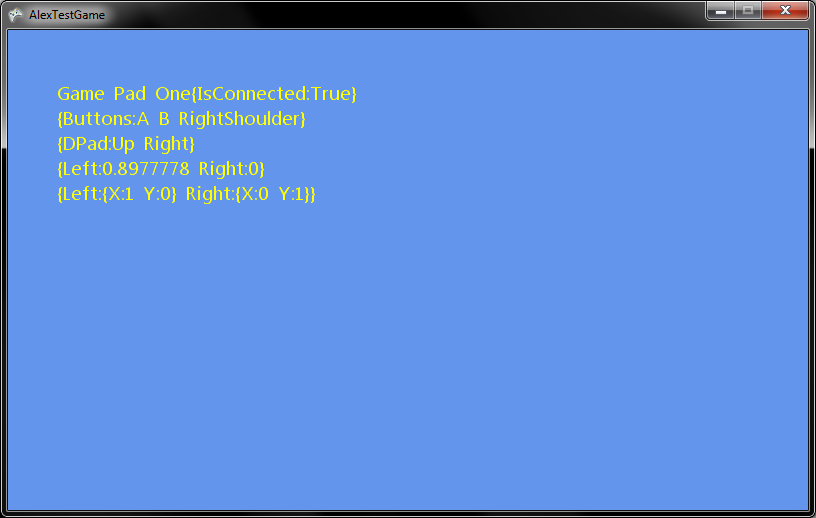


Figure 11 - Input test game

The live game output was then compared against the event that where printed out in console as shown in Figure 5 - Event output.

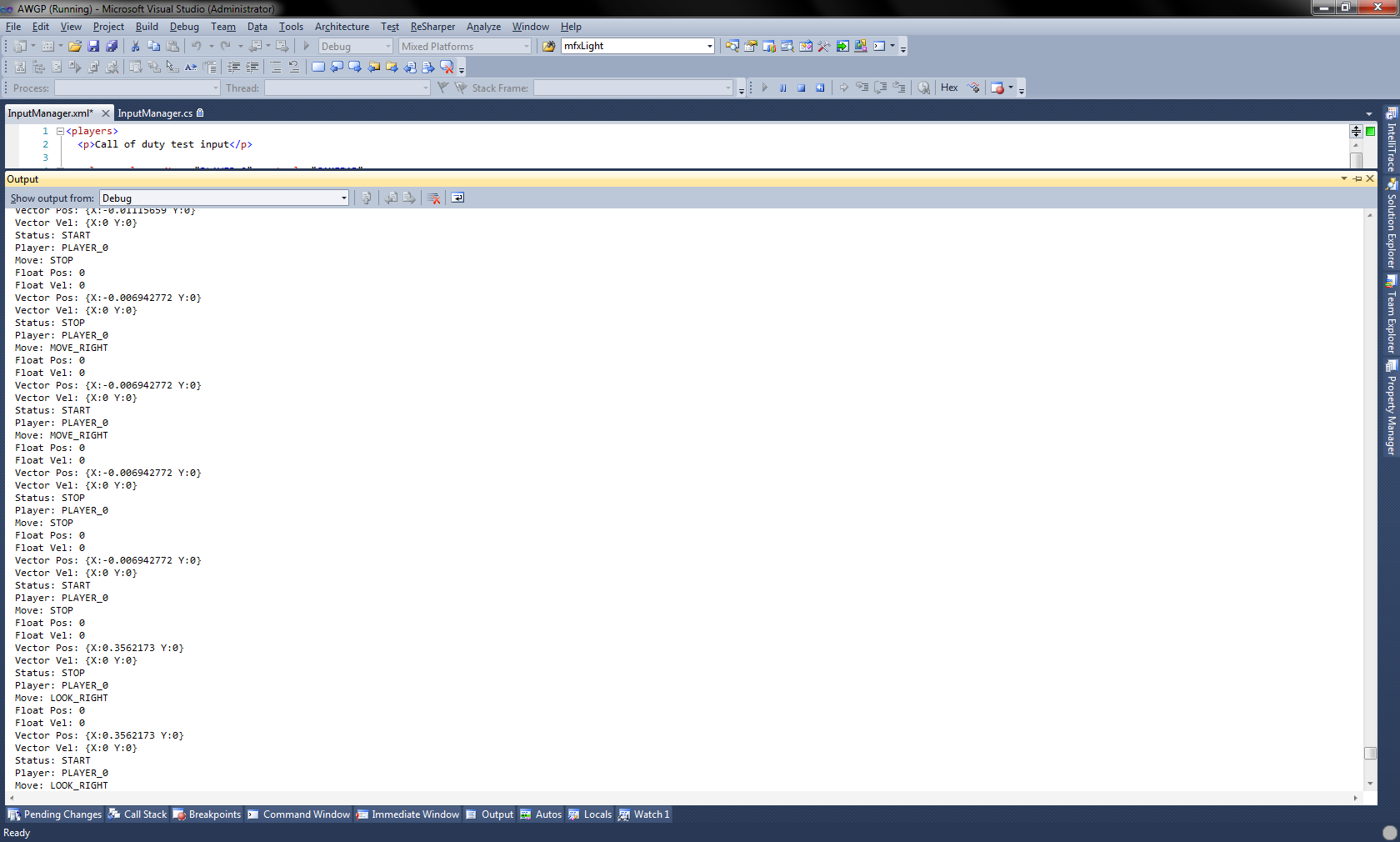


Figure 12 - Event output

Other testing was carried out such as what if the XML was not in the correct format or was not incorrect location. This resulted in try catch exceptions being thrown to the game to be caught. These exceptions then could be caught and handle by the game i.e. messages to the players, “Sorry you must setup Player One’s controls before you play the game.”

### Graphics

Despite the benefits of a robust unit testing framework, there are difficulties with applying such the technique to graphics code. Given the extremely complex nature of computer graphics, almost all modern rendering pipelines, including the component developed for this engine, make use of intermediate APIs, drivers and specialist graphics hardware in order to both ease the process of writing graphical code for the developer and to ensure that these computationally-expensive algorithms are executed efficiently. The graphical component developed for this engine, for example, used the XNA framework API, which acts as a wrapper around the Direct3D9 graphics API, which then communicates with the vendor-specific drivers in order to leverage the user’s graphics hardware and generate the final image data. It can be difficult or impossible to directly access this resultant image data in order to verify it, due to hardware and API optimisations. Additionally, graphics APIs are often very data and state-driven, with the results of API calls being very reliant on both the validity of the input data and the current state of the graphics device. These factors make writing concise and meaningful unit tests for graphics functionality very difficult, as the observable end-user results for operations are not dependent solely on the code being tested.

Though difficult, it is definitely possible to write unit tests which verify the data and calls made to the graphics API, although due to the aforementioned state-based nature of such APIs, these tests may well require more than average usage of constructs such as mock objects, test harnesses, and method stubs. The logic with this approach is the assumption that given the correct, verified data, the API will produce the correct result. However, due to driver and hardware differences, this is not necessarily the case. It should be noted that for most purposes this approach would be perfectly valid, and that platform differences are more likely to be subtle differences in quality of certain operations such as texture filtering, rather than full-blown incompatibilities or feature bugs.

Another technique is to subtly expand the scope of these unit tests into ‘functional tests’ as used by the developers of the Unity games engine (Pranckevicius, 2011). By comparing the output of tests which use various functions of your rendering pipeline to known ‘correct’ output images, you can precisely test a wide range of functionality and quickly find which platforms or code changes cause a test to fail, and how much the output differs from the desired output. In this way, this technique provides not only unit testing of indivisible features, but also a reliable method of regression testing these features as the codebase changes. This technique is very robust, and as it tests not only the code but also its integration into a larger rendering pipeline, it can be used to detect hardware compatibility issues with your rendering code. This makes it ideal for robustly testing rendering engine code, which could potentially be used in a variety of different ways by games utilising the engine. It is however expensive and time-consuming to set up, and far outside of the practical scope for this project.

Ultimately given the difficulty of effectively unit testing graphics code, as well as my inexperience with unit testing frameworks, it was decided that unit testing was not appropriate for the graphics component, mainly due to fears that it would reduce productivity on an already very restricted project schedule. Fortunately, due to the inherently visual nature of graphics code any major bugs are often immediately apparent, more so than with many other game components. Any graphical artefacts or bugs that are noticed can then be investigated using tools such as PIX, and the appropriate action corrective taken.

## Integration Testing

## Regression Testing

Regression testing was used by all members of the group. This was of high importance as lots of changes were carried out throughout the project.

# Critical Evaluation

A critical evaluation of the engine produced in comparison to the design – you should discuss critical issues related to the implementation, performance bottle necks and limitations.

[Discuss heavy use of events for inter-component communication, and the possible performance bottleneck due to that]

[Discuss limitations of graphics system; wrt performance, missing features, flexibility for games, ease of use]

[Discuss issues with the entity component design used to integrate the systems; wrt performance, missing features, flexibility for games, ease of use]

# Project Management

Explain how the project was managed from design to implementation, including the division of work amongst the group.

The section of the report explains how the project of developing a game engine was managed from design to implementation, including the division of work amongst the group members.

The project was not managed by an individual as no one individual took on the role. This meant the work was divided in a democratic or a first come first server manner. As a group we decide which components we wanted to make and what games we would like to make using the components.

We listed the items that needed to be completed, once we had established what was needed, we individually chose the component that we wanted to build. We left the games for a later meeting. Each component was developed by the individual who chose it, however rather than communicate how each component would work we chose not to. This was a strategic move to abstract each component from the others.

Daniel and Alex help Simon as the physics was the most complex component. Alex and Daniel worked on their own component without much assistance.

Once the components where near completion we started to bring them together this is where Daniel took on the role of combing the components into a working engine whereas Simon and Alex tested the components. The combing of the components was combined together by Daniel only as it seemingly was a one person job. The testing needed more attention as due to the amount of the lines of code. Refactoring and testing was the main focus for Alex and Simon.

Once Daniel had integrated the components to a point where it was possible to create games we as a group then started to plan who would create which game. Daniel and Simon worked on the racing game for testing the component together. Alex worked on the goop game while Daniel and Simon worked on the racing game. Once the racing game was one third complete Daniel stopped working on race game to test the integration. Daniel then worked on the platform game, Simon continued working on the race game and Alex on the goop game.

Alex started the report and Simon and Daniel

# Overview of the Engine

A brief overview of how the engine was utilised for each game prototype.

In this section of the report is a brief overview of how the game engine was utilised for each game prototype.

## Racing Game

## Platformer Game

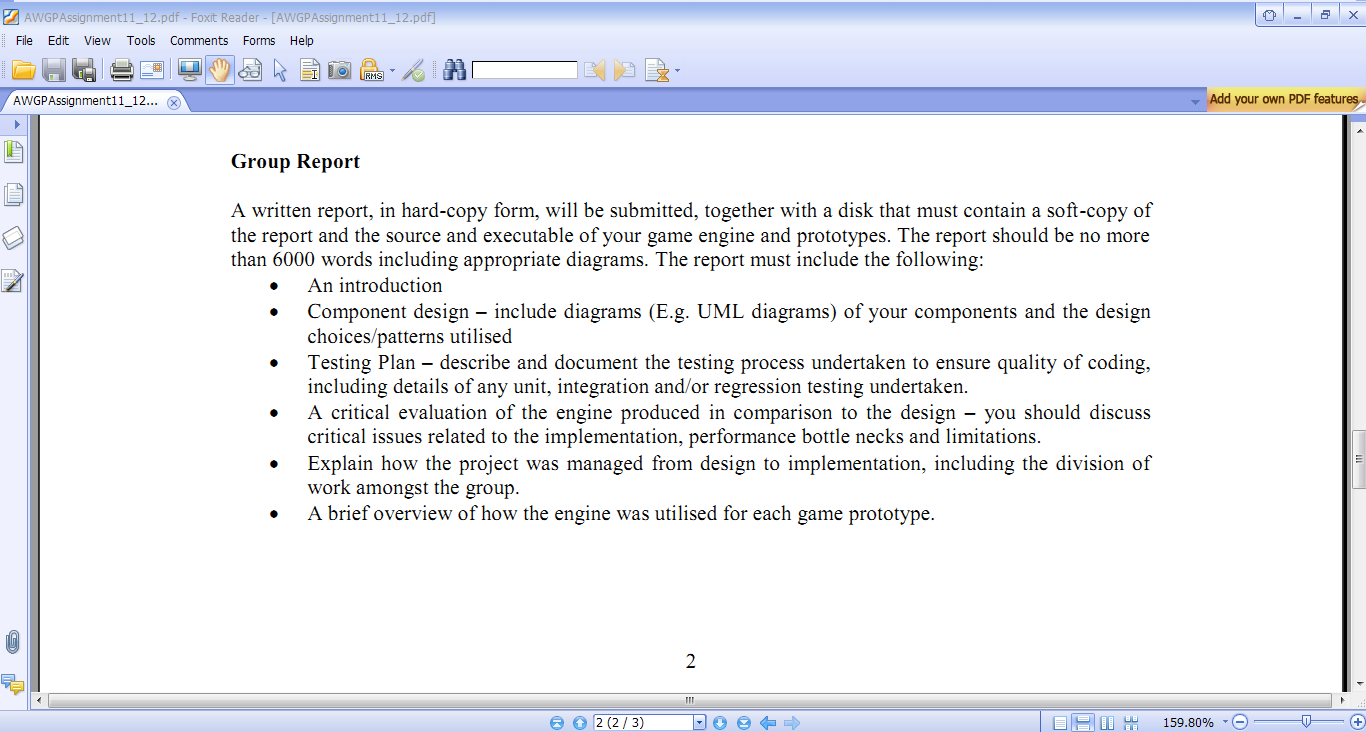


Figure 13 - Screenshot of PlatformGame

A simple platformer game was developed that would leverage the functionality of each component. Figure 8 shows a screenshot of the final game. The game features multiple player characters, dependant on the number of active input devices, which can traverse the level with physically-based motion in order to collect glowing items before the other players. Aspects of gameplay such as scoring and win or lose conditions were not implemented, as they would not help to show the functionality of the implemented game engine.

[How the input component is used & it’s functionality demonstrated]

[How the physics component is used & it’s functionality demonstrated]

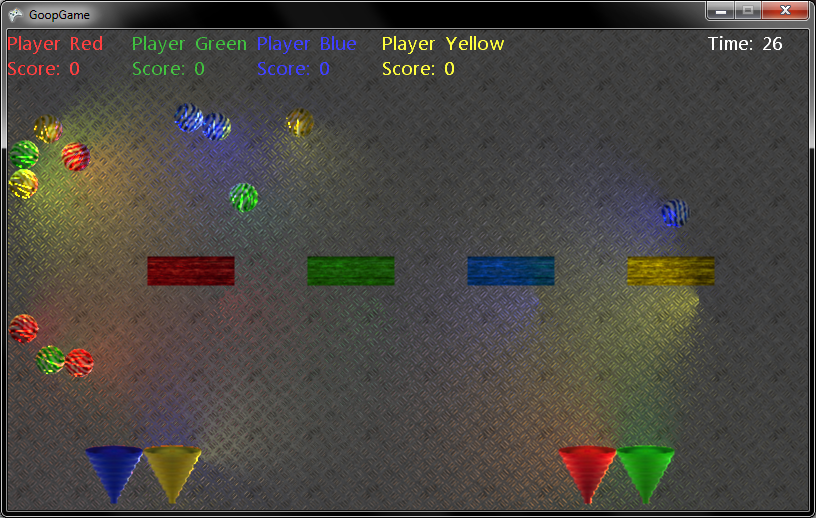
[How the graphics component is used & it’s functionality demonstrated]

## Goop Game

The goop game uses all three components that were developed. Figure 3 - Goop game shows ta screenshot from the game. All the game elements use one or all the components. The goop, player objects, goop collector, background and text all uses the graphics component to be drawn. The goop, player objects, goop collector and background have all got normal maps which give them the elution of be 3D. Also each goop, player objects and goop collector have a light each the same as its respective texture colour.

The goop, player objects and goop collector use the physics component to move in the game world. The goop rotates as they seemingly falls and it hits the side of the wall. The player objects, in conjunction with the input component move within the game world and hit the sides. The goop collectors use the physic to bounce from side to side at the bottom of the screen.

The input component is used control the player objects within the game. The game is also setup to adjust according to the number of players read in by input manager. If there are only two players profiles that are in the XML file then there is only two players in the game. The input manager allows for multiple players to play the game. The input is all abstracted from the game and only up, down, leftturn, rightturn, exit and start\_stop for the game, keyboard and gamepad can be used with this game by only changing the XML file.



Goop

Goop Collector

Player’s object

Figure 14 - Goop game

# Bibliography

Pranckevicius, A. (2011, June 17). *Testing Graphics Code, 4 years later*. Retrieved January 05, 2012, from Lost in the Triangles: http://aras-p.info/blog/2011/06/17/testing-graphics-code-4-years-later/

Shishkovtsov, O. (2005). Chapter 9. Deferred Shading in S.T.A.L.K.E.R. In *GPU Gems 2.* Addison-Wesley.

West, M. (2007, January 05). *Evolve Your Heirachy*. Retrieved January 05, 2012, from Cowboy Programming: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

# Appendices

1. Please note that the Move element would most likely be acceleration. [↑](#footnote-ref-2)
2. Please note that a full version of input UML can be found on CD, as part of the visual studio solution. [↑](#footnote-ref-3)