Computer Games Lab CMPSME22 - Coursework Report

Mech Duel Game By Adam Aldridge, John Gilbey and Thomas Linstead

1 Introduction

The project idea for our Computer Games Lab project was to create a card game based computer game. It was to be an AR (Augmented Reality) card game where real life cards are placed in view of a webcam and then the effect or characters appear on screen. The game application was created using C++ and the OpenGL libraries provided by glut. The functionality of the ARToolKit was incorporated to provide the webcam and detection features. The project as a whole was worked on as part of a team through the use of Git source control.

2 Motivation

A major goal in the creation of this game was to be able to utilise techniques and input methods which are not commonplace within the video game market. One such idea was the inclusion of an augmented reality system, giving the player the opportunity to use control methods other than that of a simple controller or mouse and keyboard.

The primary inspiration for our game was a PlayStation 3 game by the name of 'Eye of Judgment' (EoJ). EoJ uses an external card game system and adds to this this by utilising a camera setup which displays the characters on-screen during gameplay. We wanted to try and emulate this effect by creating our own card game, which could also utilise 3D model versions of the characters and have them appear in an on-screen environment. This strategy makes the game mechanics themselves more focused on the card game, whilst the software aspect plays out the gamer's actions on screen and brings their actions to life, whilst alleviating any burden of manually tracking game statistics. We also wanted to incorporate optional 3D into the game to offer a more immersive experience such as that offered by the game 'Minecraft' below.

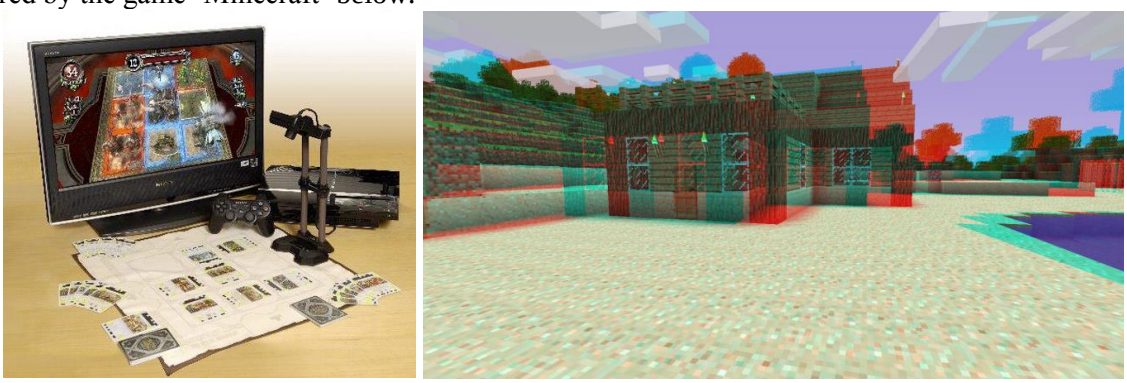
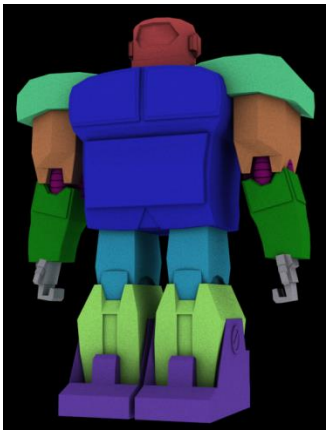


Figure 1: Left: 'Eye of Judgment' Playstation 3 game. Right: 'Minecraft'game with Anaglyph 3D enabled.

3 Modelling Characters

There were a number of 3D models created for this project including weapons, other objects and the three robot characters, Alpha, Delta and Epsilon. All the 3D models used in the game were created using 3DS Max.

Alpha



Delta

Epsilon

Various ideas modelling techniques were used to create each model:

- Alpha was the first robot to be created and was intended to be a general all-round robot, whose stats would be essentially even across the board. Alpha utilised a combination of various techniques and tools in 3DS Max, with the basic design being based off of a blueprint. The blueprint included the front view and side view of the model as 2D pictures. These were applied to planes in 3DS max and the individual sections created within the space in front so that aspects such as scale and positioning could be as close as possible. Thus, a lot of the finer shapes within the model were created by editing vertices to match some of the objects within the blueprint.
- Delta is an entirely original design which was created primarily to offset the more well-balanced Alpha robot and give a notable defence boost, whilst lowering speed in terms of stats. The point here was to create the primary sections of the body as large boxes and then edit them down so that they resembled a much larger, heavyweight-type design that still looked to be humanoid in style.
- The design for Epsilon was to promote speed with considerable power from the legs. Thus, the main body is slimmer and more streamlined than Alpha, whilst the lower legs have a considerably increased size to uphold the suggestion of leg strength, i.e. speed. This robot was created using a mix of a blueprint for some sections and some original design sections to ensure that it appeared to be built for speed.

4 MotionBuilder and MD5 Export

MotionBuilder was utilised to apply motion capture data to each model giving them various animations such as attacking and defending. This process required skeletons to be defined for each model as well as the use of bipeds within MotionBuilder. In addition, each individual component of the models had to be textured and have skin modifiers applied in order to export the model as an .md5 file, which could be used within the game itself. The processes of getting the model and components to MotionBuilder and back again to 3DS Max for .md5 export was refined over time to the point at which the process of doing this with Epsilon was relatively straightforward.

5 Model Loading

To be able to integrate animated models into our game, a file format needed to be used to read in and display these models. The .md5 file format was utilised due to the ease of the file structure. The code to load such files was extended from a base structural class with extra methods for loading in these models. This loads in the bone structure of the files and interpolates the structure per frame to create

the movement. Additional functionality of normal calculation for lighting and multi-texturing was also added to improve the quality of the visual results. Texture loading is achieved through use of the DevIL image library to allow for a wide number of image formats. Once the model loader was complete, motion capture movements were recorded so that the movements could be exported into our game. This allowed us to quickly and easily create animations for the game's characters and place them in the model loader. Extra .obj file loading classes from external source code were also integrated for any models that are loaded which do not require animation.

6 AR Implementation

Upon deciding to use AR, the first task in implementing it was to select a library to use, since implementing AR functionality from scratch would require far more time than was available for this project. After examining several open-source AR libraries such as ArUco, Goblin XNA, ARToolKit and ARToolKit Plus, ARToolKit was selected as the library that best fit the requirements of this project. This was due to its relative simplicity to work with and the large number of example applications available demonstrating how to use it. The main downside to this library was its poorly maintained state, with numerous duplicate classes and deprecated methods still in circulation within the library, making it very hard to understand and utilise some aspects of it.

Utilising AR can be summarised in two stages: firstly detecting markers, and secondly calculating the transform of those markers to render on top of them. Fortunately, the library contained example code to expedite both of these processes, causing the implementation of AR functionality to be a significantly smaller undertaking than had originally been anticipated.

6.1 Marker Detection

Markers must conform to certain rules to be detected, primarily that they are a black square with a different coloured interior, alongside the desired pattern overlaid on this interior. It is also preferable that the patterns are asymmetric and do not contain fine detail.

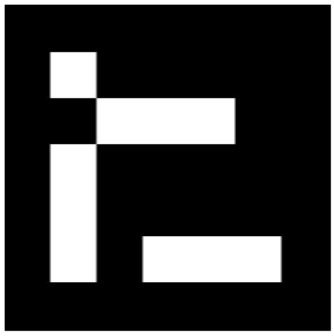


Figure 2: An example of a marker that can be detected by the application.

For this project, 54 new markers such as the one shown above were trained using the *mk_patt* application that was included with the ARToolKit. Of these, 30 were used in the final application. Marker patterns are stored and defined in a list which is readable by the *object.c* class, called *object_data2*. The list defines markers in the following format:

Comment	**Code**
Number of patterns stored:	30
Pattern name:	patt01
Pattern location:	Data/patt01
Pattern width:	80.0
Pattern centre:	0.0 0.0
Further patterns below…..	patt02

Markers are detected in the scene by scanning each video frame from the camera for black squares, storing each one found in an array, *object[]*, and then comparing each item stored in *object[]* with the list of patterns stored in *object_data2* and assigning each pairing a similarity value. If a pattern is found to match a marker in the scene with a higher similarity value than any other marker, and the similarity value is greater than 70%, then the marker is accepted and may have its associated card or 3D model rendered, assuming that it meets certain criteria set out in the *gamestate.cpp* class, such as *cost* not exceeding the active player's *manaPoints*.

6.2 Calculating Marker Transforms

In order to render the 3D model associated with a marker once it has been accepted and passed *gamestate* checks, the transform of the marker must be calculated. This is performed by the *arGetTransMat* and *arGetTransMatCont* classes provided in the ARToolKit. The availability of these classes saved a significant amount of time versus having to implement them from scratch. As well as allowing 3D models to be rendered on top of markers, the transform calculations also allow features such as robots orientating and navigating towards each other onscreen.

7 Robot Orientation and Navigation

When two robots are onscreen together, they will always rotate to face each other, regardless of the positions and orientations of the markers that they are drawn upon. This is achieved by calculating the positional differences between the two robots in the *x* and *z* axes, and then using the results to calculate the angle between the robots using the following calculation:

```
return atan2(getMarkerDiffX(), getMarkerDiffZ()) * 180 / PI;
```

The first robot is then rotated by the result in the *y* axis, and the second robot is rotated by the negative of the result in the *y* axis. If only one robot is onscreen, it will simply rotate to face forwards. The full matrix calculation can be seen below for rotating and translating the robots when attacking, this utilises the *vmath* matrix mathematics library:

```
mat = Matrix4d(gl_para);
rot = mat.createRotationAroundAxis(0, 0, getAngleBetweenRobots() + mech2Position[3]);
trans = mat.createTranslation(0, -currentStep*stepX, 0, 1);
glLoadMatrixd(Offset3D*mat*rot*trans);
```

When one robot is about to attack the other, that robot will also move towards the other whilst playing a walking animation, and back to its original location afterwards. The speed of this movement is set such that the attacking robot will always reach the other as the walk animation expires using a set number of steps. This number of steps is calculated by the following calculation:

```
int numsteps = 1.5 * gamestate::deltaTime/md5load::animSpeed;
```

Where *animSpeed* is set to 1 by default, and can be adjusted by the user in the options menu.

8 AI Implementation

When the number of players is set to 1 in the options menu, an AI is activated to take control of player 2. In this instance, at the point where player 2 would normally select their robot, instead *aiturn::init()* is called. This first selects a robot from the two not chosen by the player, then initialises the AI's card deck from all non-robot cards, then finally initialises its card hand with 2 cards drawn from the deck. Since the AI's robot is not drawn upon a detected marker, its position and rotation must be set manually. Position is set to an offset of the last known position of the player's robot. Rotations are set to 0, as the code described in Section 7 will alter its rotation to face the player's robot. This AI has two difficulty settings that can be selected in the options menu: easy and hard. The difficulty setting determines how the AI chooses what cards to play each turn. Card selection is performed by the

aiturn::chooseCard(int difficulty) method. After a card has been played, the AI moves to the attacking phase then ends its turn, returning control to the player.

8.1 Easy Difficulty

When this difficulty setting is active, each turn the AI will create a subset of its hand comprised of those cards in the hand that it has sufficient *manaPoints* to play. From this subset, it selects what card to play at random. If the subset is empty because the hand contains no playable cards, the card selection method returns -1, meaning no card has been selected and the AI should proceed to attacking.

8.2 Hard Difficulty

When this difficulty setting is active, as with easy difficulty, each turn the AI will create a subset of its hand comprised of those cards in the hand that it has sufficient *manaPoints* to play. Each card within that subset is then evaluated and assigned a score based on its properties, such as cost, type, attack and defence. To introduce an element of variety and unpredictability to the way the AI plays, each robot is assigned a personality, so that the AI will employ different tactics based upon the robot it's using. These personalities modify the way the various properties of the cards being evaluated are scored. For example, 'Delta Tank' is a robot that starts with a high number of defence points. The personality assigned to it will favour cards that further improve its defensive capabilities, by weighting the health and defence properties of cards more heavily than the attack property. Regardless of which personality is in use, the presence of a clearly-defined strategy allows the AI to pose more of a challenge to the user than the easy difficulty mode.

9 Audio

Sound was incorporated into the application by using the OpenAL library and creating methods in a new *soundeffects* class for loading in .wav files. This works by using buffers from which sounds can then be played once or set to loop such as for background music. Sounds are linked to many aspects of gameplay including animations and card activations.

10 Graphical User Interface and Menus

To be able to create a good user experience, a menu system was needed to be developed to aid with the gameplay's flow. Therefore a class was made to handle the GUI elements and the interactions between them called *menutextures*. This class contains methods for detecting which GUI element has been clicked, rendering each menu state during the program and loading in menu textures onto quads. Menu elements are rendered in Orthographic mode in front of the rest of the scene in 2D together with alpha blending to achieve an effective overlay. To further improve the quality of the menus and to incorporate dynamic stats elements, text needed to be displayed in a way that can be easily modified at run time. This was achieved by using the *freetype* library. The library is integrated together with a *freetype* class which can be instantiated to create freetype text in the menu class and positioned easily. An additional options menu was created to allow various factors of the game to be modified.

11 Gameplay and Flow

To make this application into a playable game, gameplay flow needed to be developed. For this a variety of cards needed to be designed, each with different effects and statistics modifiers. To accommodate the wide range of different functionality a database file was needed to handle the large amounts of data. A file loader was incorporated for the comma separated file format (.csv), since this format also opens in other programmes such as Microsoft Excel, it allows for easy modification of the data. This code was implemented as a method in a new static class called *gamestate*. This class stores all variables relating to the current state of the game and can be accessed by the various parts of the program that handle the several state changes of the program. For example this class handles the number of cards in the player's hand, their points and hero statistics. To handle the loaded card data, a generic '*Card'* class was also implemented which can handle any card's data. This allowed for a card list to be generated from the database for easy object orientated access to the data. To allow players to

have a system of spendable points to activate cards each turn, a random dice system was incorporated. This uses the *rand()* random number generation to obtain numbers between 0 and 1, with the amount of dice rolled increasing by 1 each after turn up to a maximum of 12.

12 Additional Elements

To refine the game a number of additional functionalities beyond the implementation of core mechanics were included. To allow for easy gameplay and to make this into a card game, 30 different card designs were created, each with different stats and designs. These are printed off with functional AR marker tags for use within the game. Shown below are four of the cards designed.

To make winning the game a more exciting experience, a particle system was developed to simulate fireworks for the final celebration. This is achieved in a *particle* class which handles the setup, advancing movement and explosion of firework points. The particles are made up of *glPoints* and arrays that handle their positions and velocities. All points of one firework start at the same place and move together adding gravity to their speeds each frame until they reach their peak, at which they start moving downwards again. Once this point has been reached, the *explode()* method is called which changes the speed of each particle, this is determined by random numbers, allowing them to go in different directions. After the firework has exploded, an alpha value is decreased per frame and once it reaches zero the firework is reset.

In addition, an anaglyph 3D mode was developed to allow the game to be played in 3D with red-cyan glasses. This works by rendering the scene twice applying a colour mask with each pass and offsetting each mask in opposite directions in the *x* axis. An example of how this is achieved is shown below:

```
glColorMask(true, false, false, false);
glTranslatef(-frustrum, 0, 0);
argDispImage(dataPtr, 0, 0);
glColorMask(false, true, true, false);
glTranslatef(frustrum, 0, 0);
argDispImage(dataPtr, 0, 0);
```

13 Testing

To test our game, we performed intensive checks ensuring that all aspects of the game worked correctly. One method employed was to perform black box dynamic testing; this allowed us to check that all cards played produced the same statistics on screen as listed on the card. This testing also checked that a game was able to be played through in its entirety. Following on from this, we performed dynamic white box testing to check the performance of the program. The methods we checked executed as expected. We also found that the frame rate of the application was limited by the maximum frame rate of the camera due to limitations within the ARToolKit library. This meant that accurate performance testing could not be executed; however the performance did not drop below the locked frame rate threshold on any of the systems tested. We finally tested the software through static white box testing to check through each other's code for any mistakes. During these tests we

performed a ‘test to succeed’ by playing the game to a winning scenario and performing attacking actions. We then also performed a ‘test to fail’ by repeating numerous in-game tasks; this uncovered a small memory leak with the audio class.

14 Conclusions

We created a unique game which has fun gameplay incorporating a selection of interesting technologies. The final game works well and covers a lot of the points in the criteria. Possible improvements to the application could include more cards being made, alongside making the gameplay flow more smoothly and intuitively without having to interrupt the card game by using the mouse for progression and confirming action prompts. In addition, any remaining memory leaks could be removed and the system optimised for possible mobile device AR implementations.

External Assets / References:

Freetype font library: http://www.freetype.org/
MD5 model loading base code: http://danielbeard.wordpress.com/2009/06/20/doom-md5-model-loader/
OBJ File loader source code: http://www.dhpoware.com/demos/glObjViewer.html
Devil Library for texture loading: http://openil.sourceforge.net/
ARToolkit Library: http://www.hitl.washington.edu/artoolkit/download/
OpenAL Library: http://distro.ibiblio.org/rootlinux/rootlinux-ports/more/freealut/freealut-1.1.0/doc/alut.html
VMath Mathematic Library: http://bartipan.net/vmath/#features
EoJ Image: http://www.vgblogger.com/wp-content/uploads/2007/09/20/The%20Eye%20of%20Judgment.jpg
Minecraft Image: https://hostr.co/file/xFdQucA/2011-03-18_13.06.37.png

List of Work

Adam Aldridge (Agreed Contribution 25%):

- Investigated AR with goal of determining most suitable library (ARToolKit);
- Trained 54 markers;
- Implemented robot orientation and in-game navigation;
- Motion capture;
- AI (Random-choice & personality-driven);
- Integrated AI (with Toms help).

John Gilbey (Agreed Contribution 35%):

- 3D modelling;
    - Robots: Robot A (Alpha), Robot D (Delta), Robot E (Epsilon);
    - Weapons: Sword 1 (unused), Sword 2, Sword 3, Broadsword, Gun (unused);
    - Other: Wings, Dice, Coin (unused).
- Character rigging;
- Assigning/plotting animation to models;
- MD5 export;
- Motion capture.

Thomas Linstead (Agreed Contribution 40%):

- MD5 model loading with animations;
- Implemented external OBJ model loading code for static models;
- OpenAL audio implementation in ‘soundeffect’ class;
- Basic AR integration with main project;
- GUI and menu implementation in ‘menutextures’ class;
- ‘Freetype’ font library support with custom font;
- Card database in .csv format & data loading;
- Generic ‘Card’ object class to handle card data;
- ‘DevIL’ texture library incorporation;
- Gameplay and flow through step by step phases and turns;
- Implemented ‘gamestate’ static class to handle the current state of the game;
- Motion capture;
- Created a dice rolling system for spendable points;
- Designed and created 30 cards for use in gameplay;
- Robot orientation matrix mathematics utilising the ‘vmath’ mathematics library;
- Implemented particle system for fireworks on winning screen;
- Anaglyph 3D support;
- Large selection of modifiable options for options menu;
- Helped to integrate the AI system.

User Manual: Mech Duel

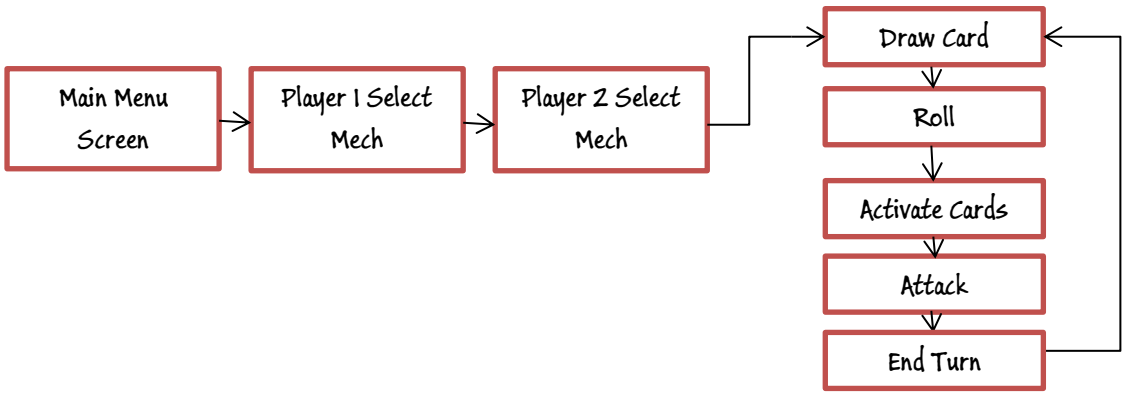
Goal: To beat the other player's robot by reducing their health (hearts) to zero.

Controls:

- Mouse Left Click - Click on menu buttons to progress gameplay.
- 'P' Keyboard Button – Options menu with modifiable options.
- 'Space' Keyboard Button – Toggle Anaglyph 3D mode.
- 'Escape' Keyboard Button – Exit game.
- Place cards in view of the camera to activate during your main turn.

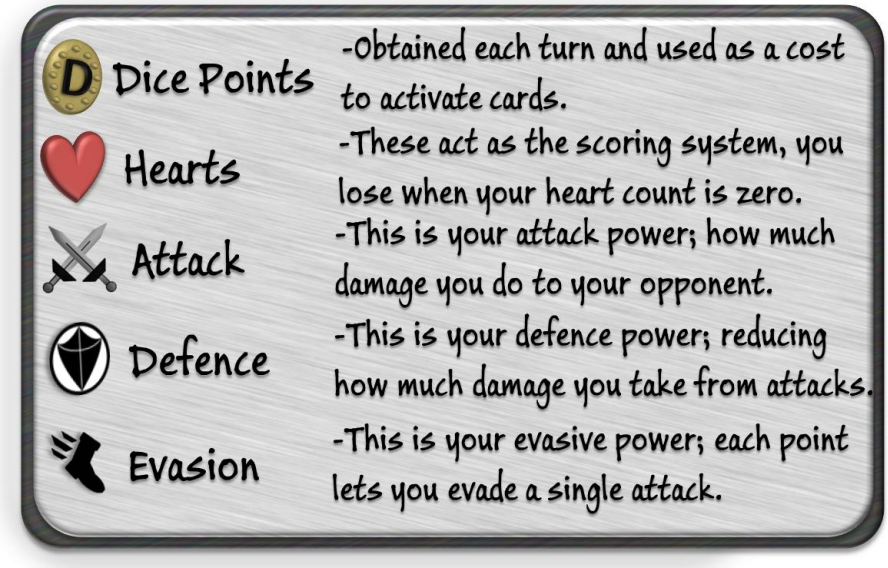
Gameplay:

Each player selects a robot to use for the game and presents them to the webcam when told to. Each player starts with a hand of 3 cards. Turns then progress with each player drawing a card, rolling dice to earn dice points which in turn are used to activate cards from your hand and then robots attacking each other.



Statistic-based Gameplay:

The gameplay revolves around point building to gain an advantage over your opponent. The main statistics are detailed below.



System Requirements/Equipment:

- Windows PC which is OpenGL compatible.
- Good quality webcam which can be pointed downwards towards a desk.
- Deck of compatible AR cards.
- Anaglyph red-cyan 3D glasses for 3D mode. (Optional)