Unnatural Gaming

Checkmate

Requirements Document

Revision #: 03
Revision Date: March 3rd, 2003

Table of Contents

1. Background

Checkmate is an online team deathmatch game incorporating elements of Chess. Two teams will be pitted against each other for the sole purpose of eliminating the opposition's King. This document will show that we have a good handle on the requirements and that we understand how the system will work. This also shows how we are handling the user interface and how we are organizing the high-level architecture.

2. Requirements

2.1. Non-Functional Requirements

1. Checkmate will be developed in 'UScript' using *Wotgreal* editor.
2. Builds will be managed using *ant scripts*.
3. Bug tracking and reporting will be managed using *Bugzilla*.
4. Version control will be managed using *CVS*.
5. Minimum system requirements for the game will be the same as the requirements for UT2003:
    - P3/Athlon 733MHz
    - 128 MB Ram
    - 3 GB Hard Drive Space
    - 16 MB TNT2 class video card (DirectX 6 compliant)
    - TCP/IP connection
    - Windows 98/ME/2000/XP
6. SSH login for external access to *CVS*.
7. The expected performance will be equal to that of UT2003.

2.2. Functional Requirements

8. Able to play with other Checkmate users over the Internet, or a local area network
9. There are 2 teams for a player to choose from. The teams compete against each other to try and defeat the other's King.
10. Each team will have 1 King – no more, no less. See requirements 19 & 22 for more information.
11. There will be any number of non-King Player Classes on a team
12. Player Classes will be unique (See Appendix A). Each class will require the player to adopt a different play strategy, based on the Player Class characteristics (movement speed, weapon characteristics, player health, special skills, etc).
13. Point values per class:
    - Pawn: 1
    - Knight: 3
    - Bishop: 3
    - Rook: 5
    - Queen: 9
    - King: 10
14. Each Player Class must have an overall advantage to the classes below it (point-wise). There may be general advantages or disadvantages to a class based on game situations, but overall the higher-value Player Classes should be more powerful than lesser-value Player Classes
15. Knight and Bishop Player Classes should be balanced with each other. Both have strengths and weaknesses in certain game situations, however overall they should be of roughly equal power.
16. A team scores a point by eliminating the other team's King

17. The King's health will scale based on the number of players on the opposing team. If more players join the opposing team, the King's health will scale up accordingly, and if there are less players on the opposing team, the King's health will scale down accordingly.
18. A match ends by reaching a certain team-score limit and/or time limit (designated by the server administrator).
19. Each team's King for a new round is determined by the highest scoring players for each team. The exception to this rule is that if the highest scoring player already played a full round as King, they will not be eligible to be the King for a second consecutive round.
20. If a player who is controlling the King leaves the team, the highest-scoring player of that team becomes promoted to King. If no replacement is available (if there are no players on the team), it is considered a forfeit and the other team scores
21. Players are awarded credits based on their current class and the class of the enemy they fragged, according to **Equation 1**.
22. The King (and only the King) will be played in third-person[1] view.
23. Player Class selection through manual menu entry, or menu popup upon death
24. There will be a radar:
    - King will see all his team mates.
    - All other Player Classes will only see the direction their King is (relative to themselves)
25. King will be able to give voice commands (command menu)
    - All other Player Classes will be able to respond (response menu)

3. Example Data and Scenarios

For the scope of testing and verification, Checkmate does not depend on external, user-provided data. The user is required to enter data such as their name and such, but this does not affect the behaviour of the system. Rather, the testing verification is based on user interactions. Although we will use scripted players (bots) to aid in some aspects of testing and verification, we will primarily test the system with human-controlled players. The critical test scenarios are outlined in the following section (detailed use case model).

[1] Typically, every player will playing the through the eyes of their character, however the King will be played from a third-person perspective, where the viewing camera is just above and behind their character

4. Detailed Use Case Model

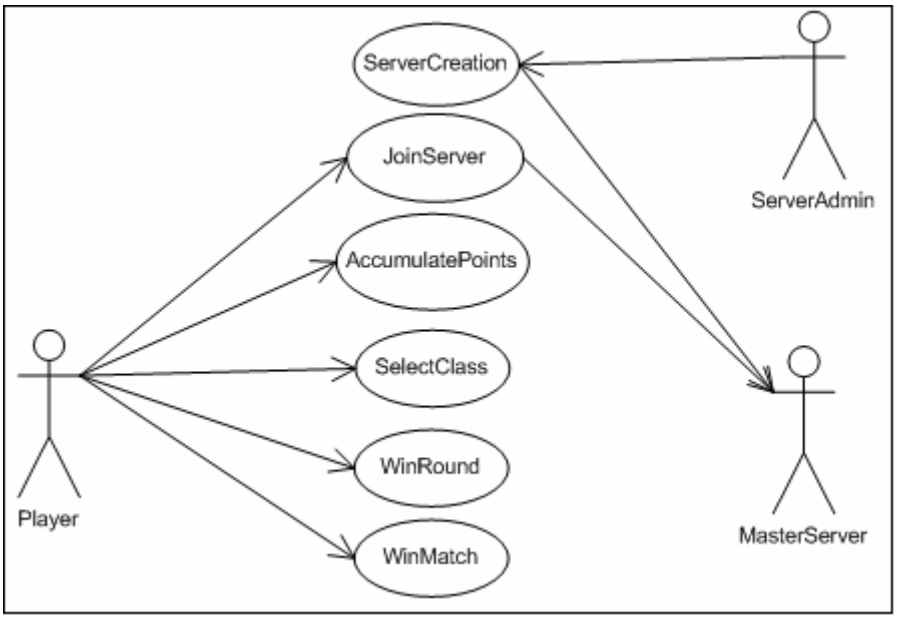


Figure 1: Use case diagram

1. Server creation

Use Case Name: ServerCreation
Requirements Addressed: #8
Actor initiating this Use Case: ServerAdmin
Actors Participating in this Use Case: ServerAdmin, MasterServer
Goal: To start a server running Checkmate, to which other Players can join
Preconditions: UT2003 & Checkmate are installed, internet connection is available
Normal flow:
1.1. Server administrator chooses server settings, and starts server
1.2. Master server is notified of the new server, and updates master list
Exceptions:
1.1. If no internet connection is available, the server will only be seen on the list of local area network games
Scenario:
1. Server *"Killa Checkmate Server"* starts up.
    1.1. The server administrator selects the initial map and starts the game server.
    1.2. The master server is notified of the new server and updates its server list.
Postcondition: Checkmate server is running, and is listed on the master server list

2. Players join a server

Use Case Name: JoinServer
Requirements Addressed: #8, #9, #10, #11, #23
Actor initiating this Use Case: Player
Actors Participating in this Use Case: Player, ServerAdmin, MasterServer
Goal: To populate the server with players, so that they can play Checkmate together
Preconditions: Server has been created, Player has an internet or local area network connection to connect to server
Normal flow:

2.1. Player chooses a server to join
2.2. Player's CD-key is authenticated by the master server
2.3. Player joins server
2.4. Player selects a team to play on
Exceptions:
2.2. Player does not have a valid CD key for authorization, and is denied server entry
2.3. The server has already reached the maximum number of player slots available, so the player receives the message "Server is full"
Scenario:
1. Player *"Neo"* joins the server.
    1.1. The list of available servers is obtained from the master server; *Neo* selects *"Killa Checkmate Server"* from the list of servers.
    1.2. The master server authenticates the client and allows the connection.
    1.3. *Neo* selects a team to join, he chooses the Black team.
    1.4. He is assigned a starting player class.
        2.1.4.1. Since the Black team is currently empty, he spawns in as the Black King.
2. Player *"Morpheus"* joins the server.
    2.1. Repeat 2.1.1. & 2.1.2.
    2.2. *Morpheus* selects the White team.
    2.3. He is assigned a starting player class.
        2.2.3.1. Since the White team is currently empty, he spawns in as the White King.
3. Player *"Trinity"* joins the server.
    3.1. Repeat 2.1.1. & 2.1.2.
    3.2. *Trinity* selects the Black team.
    3.3. She is assigned a starting player class.
        3.3.1. Since the Black team already has a King, she spawns in as a Pawn.
4. Player *"Cypher"* joins the server.
    4.1. Repeat 2.1.1. & 2.1.2.
    4.2. *Cypher* selects the White team.
    4.3. He is assigned a starting player class.
        4.3.1. Since the White team already has a King, he spawns in as a Pawn.
5. Player *"Switch"* joins the server.
    5.1. Repeat 2.1.1. & 2.1.2.
    5.2. *Switch* selects the Black team.
    5.3. She is assigned a starting player class.
        5.3.1. Since the Black team already has a King, she spawns in as a Pawn.
Postcondition: Players have joined the server, and are playing the game together

3. Players accumulate points

Use Case Name: AccumulatePoints
Requirements Addressed: #13, #21
Actor initiating this Use Case: Player
Actors Participating in this Use Case: Player
Goal: For each player to frag members of the opposing team, in order to accumulate points ("points" being a general term to describe kills, score and credits)
Preconditions: Multiple Players have connected to a server
Normal flow:
3.1. A Player frags a member of the opposing team by shooting them
3.2. If the victim's Player Class was greater than or equal to that of the Player who got the frag, credits will be awarded according to **Equation 1**, and that number will be added to their

score ("score" being a value equal to the total number of credits accumulated)

3.3. Regardless of player class, the Player who got the frag will receive another "kill" beside their name on the scoreboard

Exceptions:

3.1. If a Player shoots an opposing Player, but not enough damage is done to bring their health to 0, they will take damage, but no points will be awarded

Scenario:

1. *Trinity* (Pawn) finds *Cypher* (Pawn) and frags him by shooting him with her weapon.
    1.1. *Trinity* is awarded one credit for fragging a player of equal class, score and kills are also incremented
2. *Part 1.* is repeated another two times.
    2.1. *Trinity* now has three points banked.
3. *Morpheus* (King) frags *Trinity* (Pawn) by punching her with his fists.
    3.1. *Morpheus* is not awarded any points for fragging a player of lesser class.

Postconditions: Kills, Score, and Credits will be incremented for those who have fragged players of greater or equal Player Class.

4. Players select player classes

Use Case Name: SelectClass

Requirements Addressed: #11, #12, #13, #14, #21

Actor initiating this Use Case: Player

Actors Participating in this Use Case: Player

Goal: For a player to choose a new Player Class to play as

Preconditions: Player has accumulated enough credits to select another Player Class

Normal flow:

4.1. Player dies

4.2. Class Menu appears, offering the chance to respawn as a different class

4.3. Player selects a new class

4.4. Player respawns as the new Player Class

4.5. Credits are subtracted from their credit total, according to the Player Class value

Variation:

4.6. Player brings up Player Class selection menu (while "alive" in the game)

4.7. Player selects a new class

4.8. Upon death, Player respawns as the new Player Class

4.9. Credits are subtracted from their credit total, according to the Player Class value

Exceptions:

4.3. & 4.7. If Player attempts to select a Player Class that costs more than they can afford (in credits), they will not be allowed to select that class

Scenario:

1. *Trinity* selects a player class to spawn in as.
    1.1. *Trinity* has three credits banked, so she has enough to choose Knight (3 credits), Bishop (3 credits), or Pawn (free); a menu pops up upon her death and she selects Knight.
    1.2. *Trinity* respawns as a Knight, and three credits are removed from her banked credits.
2. *Trinity* (Knight) frags *Cypher* (Pawn).
    2.1. *Trinity* is not awarded any credits for fragging a player of lesser class.
3. *Cypher* respawns.
    3.1. *Cypher* has zero credits, so he can only afford to be a Pawn (free); the class selection menu is not displayed upon his death.
    3.2. *Cypher* respawns as a Pawn.

Postconditions: Upon death, Player respawns as the Player Class they selected, and the appropriate number of credits are subtracted from their credit total.

5. A member of one team frags the other team's King, and a new round begins

Use Case Name: WinRound
Requirements Addressed: #10, #13, #15, #18, #21
Actor initiating this Use Case: Player
Actors Participating in this Use Case: Player
Goal: For a player from one team to frag the opposing team's king, thus winning the round for their team, and causing a new round to begin
Preconditions: There is at least 1 player on each team
Normal flow:
5.1. Player from one team frags opposing team's King
5.2. The appropriate amount of points are given to the Player to be added to their score, and their kills number is incremented
5.3. The Player's team score is incremented
5.4. New Kings are assigned for each team, based on who is the highest scoring Player for each team (with the exception that a Player cannot be King twice in a row)
5.5. All other players respawn as Pawns
Exceptions:
5.4. If the server settings dictate that the match ends after a team has accumulated a certain number of victories, then the match has ended (which falls under Use Case 6, below)
Scenario:
1. *Trinity* (Black Knight) frags *Morpheus* (White King).
    1.1. According to **Equation 1**, *Trinity* is awarded four points $\left(ceiling\left(\frac{10}{3}\right)\right)$.
2. Since the White King was eliminated, Black's team-score is incremented; the team scores are now Black 1, White 0; this is the end of the first round.
3. New Kings are assigned for both teams; this is based on the highest scoring (accumulated points) player.
    3.1. On the Black team, *Trinity* has a score of seven and *Switch* has a score of zero, so *Trinity* respawns as the Black King for the new round and her score is reset to zero.
    3.2. On the White team, *Cypher* is the only non-King player, he respawns as the White King for the new round and his score is reset to zero.
4. All other players respawn as Pawns.
5. All players' credits are reset to 0 (score and kills stay the same)

Postconditions: The Player who fragged the opposing team's King is rewarded (score and kill values), the winning team's score is incremented, and a new round begins (with new Kings for each team)

6. The match ends

Use Case Name: WinMatch
Requirements Addressed: #10, #13, #15, #17, #21
Actor initiating this Use Case: Player
Actors Participating in this Use Case: Player
Goal: For a player from one team to frag the opposing team's king, thus winning the round for their team, and putting their team's score at the match limit (which wins the match for their team)
Preconditions: At least 1 player on each team, a team is one round below the round limit that has been set for the match (round limit set by server administrator upon server creation)

Normal flow:
6.1. Player from one team frags opposing team's King
6.2. The appropriate amount of points are given to the Player to be added to their score, and their kills number is incremented
6.3. The Player's team score is incremented
6.4. The round limit for the match has been reached, and the Player's team is declared the victor for the match
6.5. The server goes to the next map in the cycle, and all players load the next map
Exceptions:
6.5. If a Player does not have the next map in the server's map list, it will be automatically downloaded (if the server admin leaves this feature on)
Scenario:
1. During a match, use cases 2-5 may reoccur indefinitely.
2. A match ends after a certain amount of time, or after a certain number of rounds have been played, or after a team reaches a certain score (these variables are determined by the server administrator).
    2.1. The team with the highest score at the end of the match is declared the winner.
    2.2. A new map is loaded, a new match starts, team scores and individual points are reset.
Postconditions: Once the match has ended and a team has been declared the victor, the next map in the cycle is loaded.

5. Non-Functional Features

Quick Response Time
Checkmate will be a fast-paced, real-time FPS, rather than a turn-based game like chess, therefore quick response time is an important feature of our system. As mentioned previously, Checkmate is a modification of the Unreal Tournament 2003 system, which already has efficient network code in place. From the user's perspective, all inputs and outputs will be processed within a negligible amount of time (for example, if the user clicks a mouse button to fire their weapon, there should be no noticeable delay between their input request, and the processing of that request on the screen). Since we will be adhering to similar coding principles as the original system, which has excellent response time, there should be no delays.
There are only two exceptions to this rule: 1) if the user is running the bare-minimum system requirements, the video may suffer small amounts of "choppiness", and 2) if the user is playing in a server with a lot of latency, there will be some delay in response time. Both of these cases are common to all first-person games, and are well known to everyone in our target demographic. For testing this feature, the main concern is that the response time is quick when playing over the internet with a cable connection on a fast server. This is the setup most used by our target demographic.

Peak Workload Handling
The maximum number of Players that can be on a given Checkmate server at the same time is 32. Therefore, this will be the peak value we will be required to test and support. The 32 Player limit is a fairly common one among first-person multiplayer games, so from a design point of view we will be spending some time to make sure that we aren't sending too much data over the network (internet or local area). To test this peak value, we will be using bots (computer controlled players), as well as human players.

Scalability

The game is designed such that it can be played with any number of players on each team. The number of players on a server is dynamic (players can enter or exit the game at any point), so to increase scalability, the King's health will scale up or down based on how many players are on the opposing team. This also helps balance the gameplay if there is a mismatch of players on each team.

Usability
The user interface for Checkmate has been designed using proper UI guidelines, therefore it should be easy for a novice to understand and use. However, it is important to note that our classification of a "novice" user is someone who has some experience playing first-person shooter games, not just any average person. This is a safe assumption to make, since our modification will only be available for download over the internet, therefore our "novice" user will be someone who already has UT2003, and at least understands the concept of a game modification. To test this feature, we will be using video-tape analysis on a novice user.

6. High Level Architecture

6.1. Deployment

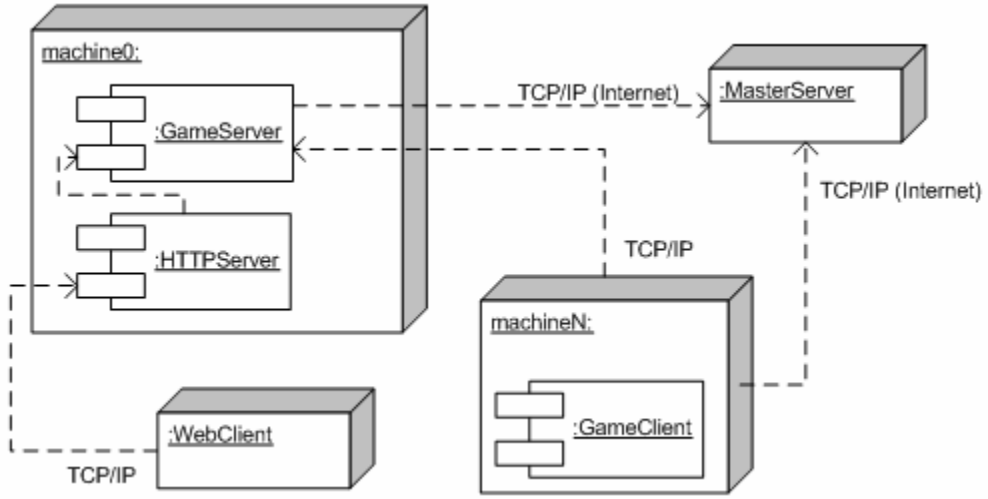


Figure 2: Deployment of Checkmate

Modified components:
GameServer: The authoritive component that maintains the true instance of the game and serves the clients.

GameClient: Runs an interpretation of the game on its own process. The client runs a proxy version of the game which attempts to simulate and predict what is actually happening on the server.

The GameServer and GameClient both run their own instance of the same executable, but as different roles. (authoritive vs. proxy). The authoritive instance regularly updates the proxies with the game state. These updates are limited by bandwidth, so a dial-up connection receives updates less frequently than a LAN connection. The server must also maintain and prioritize what is most relevant to the clients, so that when bandwidth is limited it can send what is most important.

Unmodified component:
HTTPServer: The machine running the GameServer can also run an HTTPServer that allows remote administration through a web client. We do not plan to modify this component.

External components:
WebClient: Any HTTP client, used to connect to the HTTPServer for administration.

MasterServer: Server(s) hosted by Epic Games, which list the instances of the GameServers available via the internet. The MasterServer also validates the clients that attempt to connect to the GameServers to ensure that they have a valid license.

6.2. Packages

We will produce a single library file, **Checkmate.u**. Therefore, there will be one package which is shared by both the GameServer and GameClient. A 'U' file is similar to a DLL file, but it contains UScript byte-code so it is valid for any platform that can run UT2003 (Windows and Linux).
The only reason to distribute multiple libraries for a UT2003 modification is if the author wishes to package reusable components. We do not plan to distribute reusable mod components, so it makes more sense to package Checkmate in a single library. The Checkmate library contains classes and other resources such as textures, sounds, and static meshes (3D objects).
Figure y shows the packages that Checkmate depends on. Let's take a brief look at what these packages are used for and some of their important classes.

Core contains the fundamental classes of UScript. Of particular importance, Core contains the class Object, which all other classes are derived from.

Engine contains the core definitions of a UT2003 game. Class *Actor* describes the basic behaviour of UScript objects. The various player classes of Checkmate are subclasses of *Pawn* which describes the behavior of objects that can be controlled by a player or a bot.

Gameplay contains the classes that define player interactions and environmental effects.

UnrealGame contains the generic outlines of various game types. *Checkmate* is a subclass of *TeamGame* from this package which defines the basic rules of a team-oriented game.

XInterface contains the user interface components. This package will be used to develop Checkmate's user interface.

XWeapons contains the weapons available in UT2003. This package is only relevant for weapons in Checkmate that behave similarly.

XEffects contains the classes that describe various visual effects such as explosions and bullet tracers.

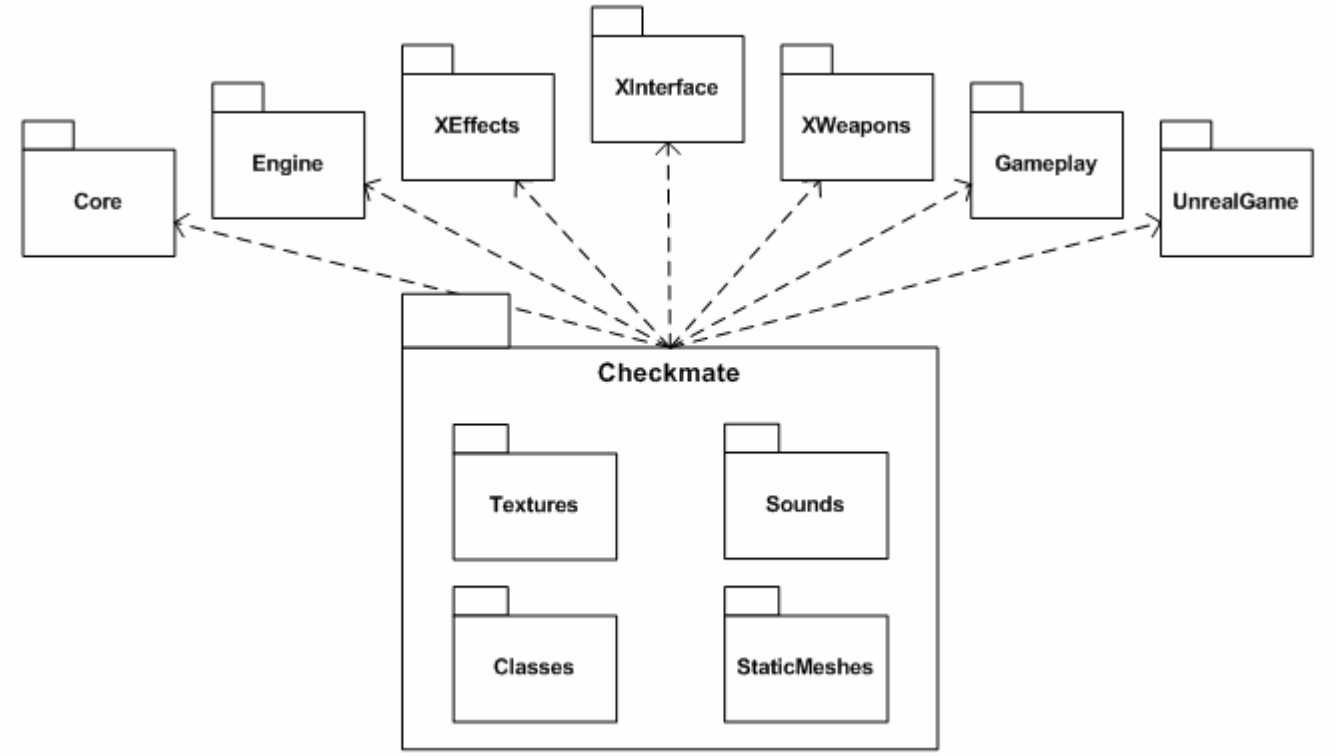
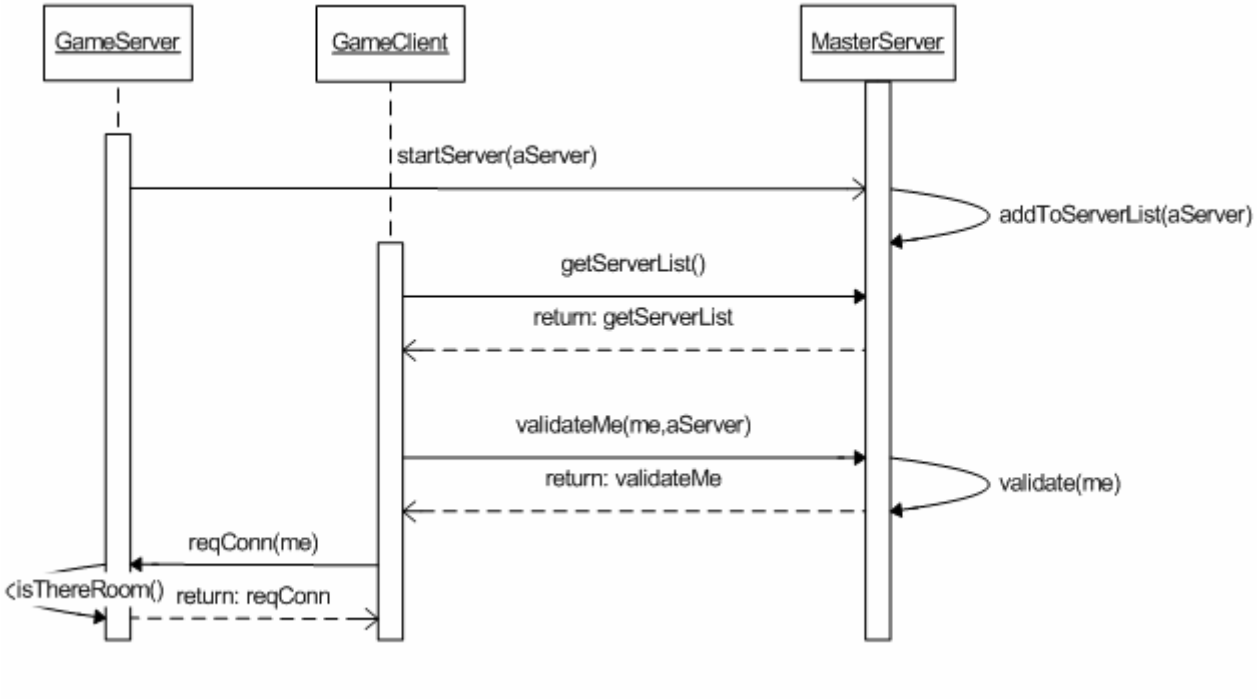


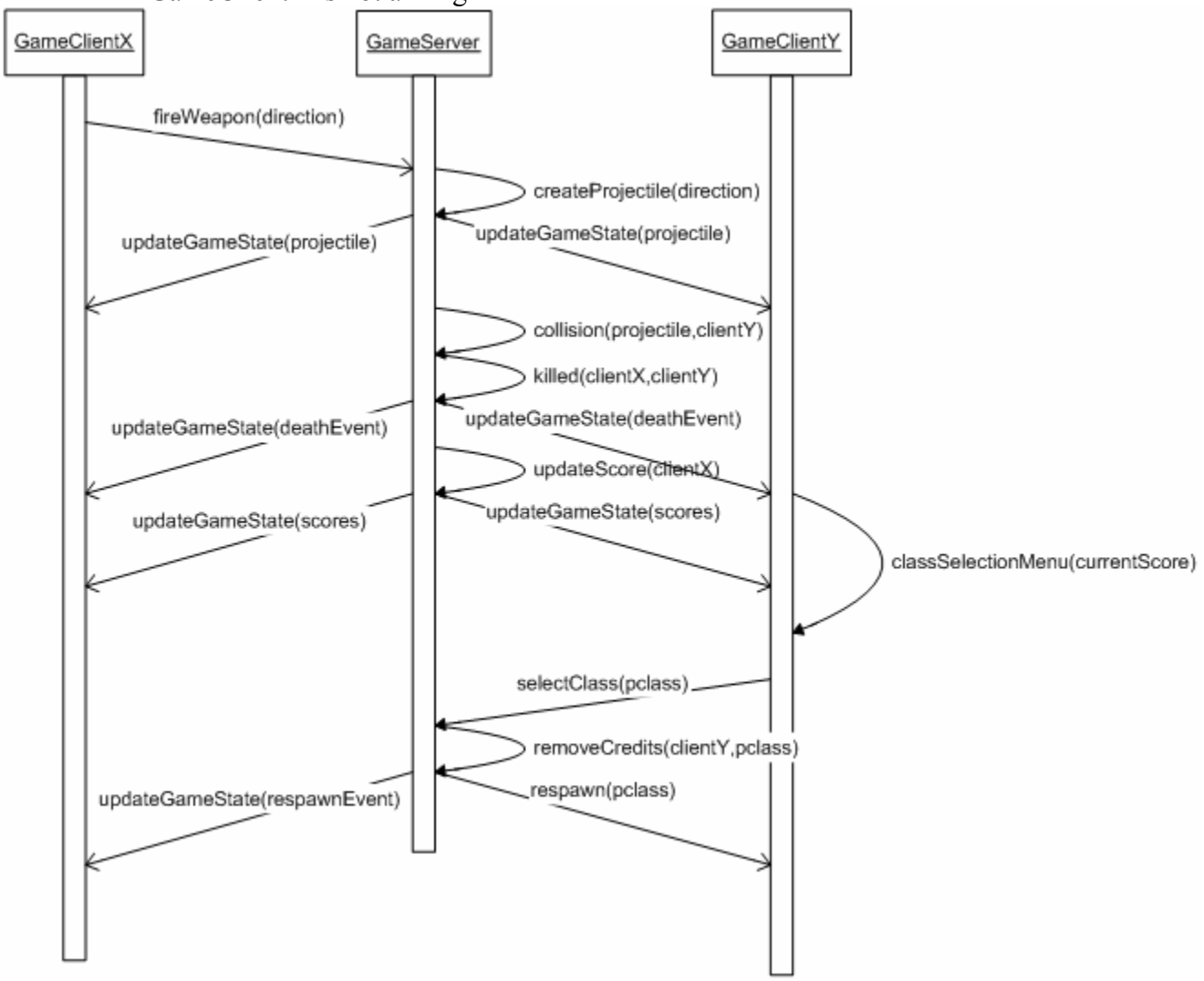
Figure 3: Package dependencies of Checkmate

6.3. Component Interactions

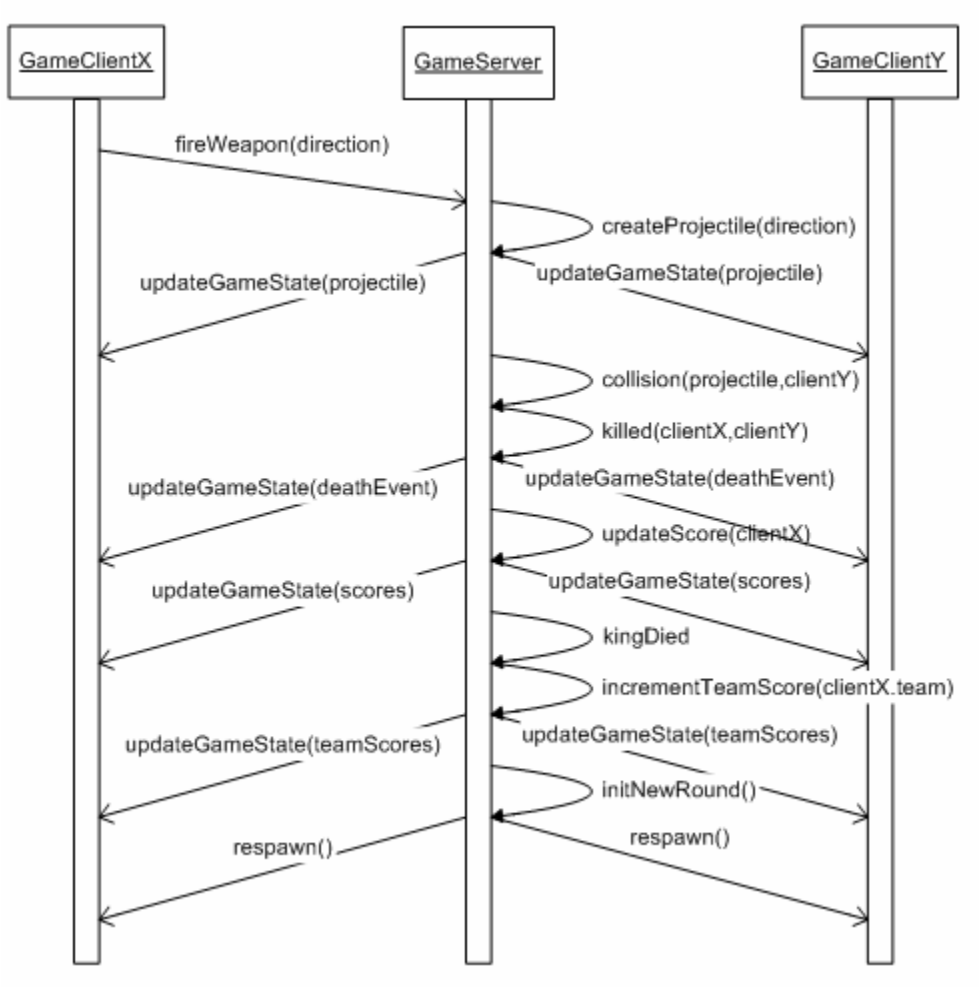
Interaction 1: GameServer initialization and GameClient connection
Referenced use cases: 1, 2
Constraints: - GameClient has a valid game license



Interaction 2: Accumulating points and selecting a player class.
Referenced use cases: 3, 4
Constraints: - GameClientX and GameClientY are connected and on different teams
- GameClientY is not a King



Interaction 3: Winning a round.
Referenced use cases: 5
Constraints: - GameClientX and GameClientY are connected and on different teams
- GameClientY is a King.



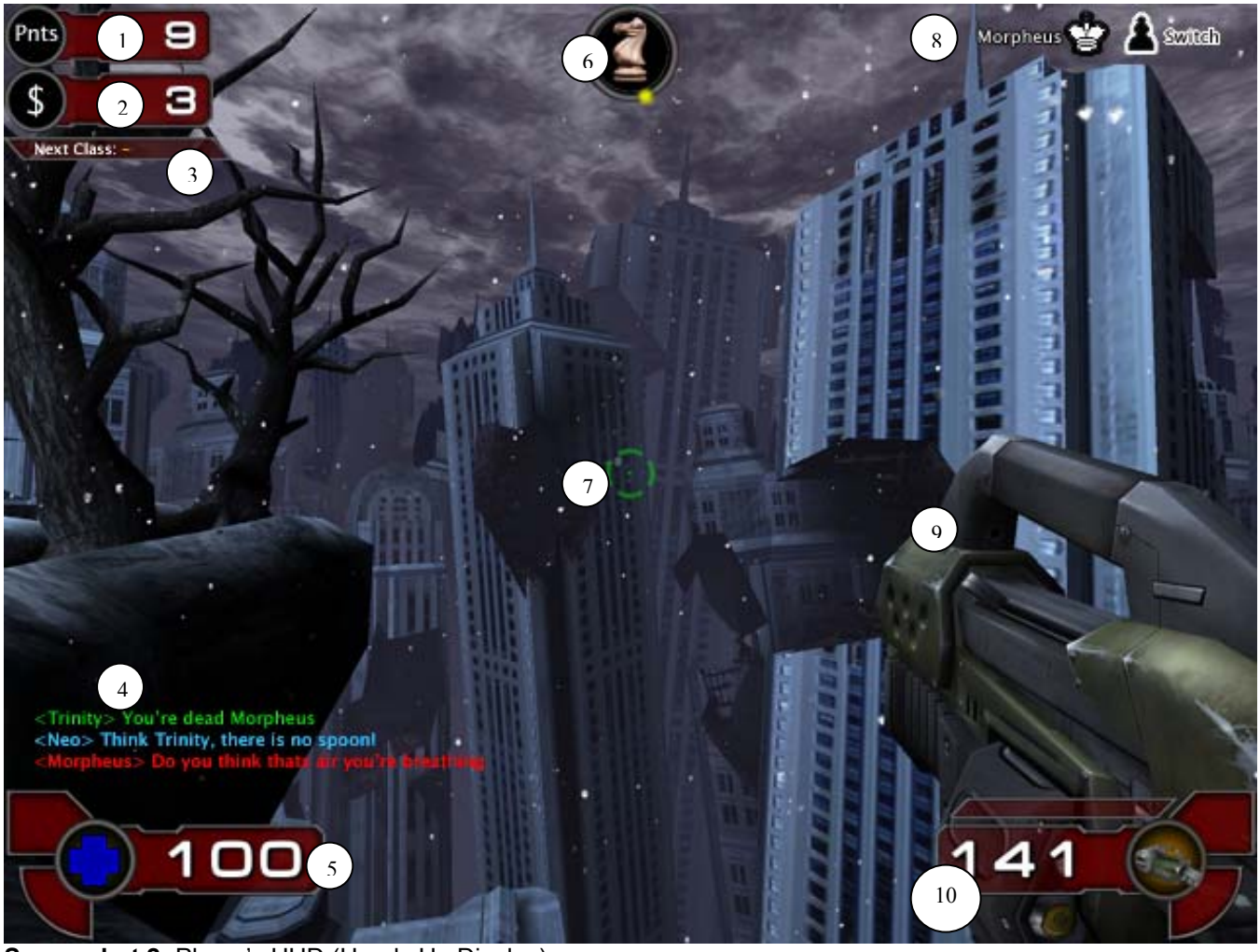
7. User Interface Mockup

Notes about these screenshots:

1- These screenshots were edited in Photoshop. This is what we would like the final version to look like, however the demo version will not look like this.
2- As mentioned in the Project Definition we are only modifying the in-game UI.
3- The images are in 640x480 resolution.
4- Quality of text is poor because the images have been scaled down to fit in the document

Screenshot 1: King's HUD (Heads Up Display)

1- These are the King's command menus. From these menus the King will be able to issue commands such as attack, defend, target etc.
2- King's Health.
3- This is the King. As noted the King will be the only player viewed in third-person.
4- King's radar. From here the King can see the distance from himself to his teammates



Screenshot 2: Player's HUD (Heads Up Display)

1- Player's score. The score is the accumulated amount of credits. This is also the value used to rank the player.
2- Player's credits. This is used to obtain new classes.
3- If a player has chosen a class for their next life it would be noted here. In this case the player has not chosen a class yet so a '-' is shown.
4- These are chat messages. Format is '<name> message'.
    i. Green text is for message relayed to everyone from a teammate.
    ii. Blue text is for team messages.
    iii. Red text is for messages broadcasted from the enemy
5- Player's Health.
6- Player's class (in this example, a knight). The yellow circle indicates the direction of the team's King (in this case the King is directed behind the Player).
7- Crosshair.
8- These are death messages. When a player frags another player, a message is sent to everyone on the server. The message format is:
    i. <killer's name> <killer's class icon> <victim's class icon> <victim's name>
    ii. The message is also color coded with the team colors.
    iii. In this example: the White King Morpheus fragged the Black Pawn Switch
9- Player's weapon.
10- Amount of ammo left.

Screenshot 3: Class Selection Menu

1- Values 1-5 used for keyboard entry of desired class.
2- Orange highlight for where the cursor resides on the menu. The cursor can be from keyboard input (arrow keys) or from a mouse cursor.
3- Classes that cannot be obtained (not enough credits) are grayed out and are unselectable.
4- Class information. Format is <icon> <Class Name>
5- Credits needed to obtain class.
6- Credits available to the Player.

Screenshot 4: Scoreboard

1- Time remaining in the current match.
2- Team information section.
3- Number of players on team
4- Team Score. In this case the Black team has 1 point and the White team has 0 points.
5- Score limit for this match.
6- Class icon column. Players will be able to see the classes for all their teammates and the King of the opposing team, but not of the other players of the opposing team.
7- Player Name column. All the players on the server are listed here.
8- King is always on top of the score board. His health will be showing to everyone. In this case Neo has a full bar of health and Morpheus has about a third left.
9- A highlight is shown for the Player's position in the scoreboard. In this case Trinity is the player looking at his/her scoreboard.
10- Score column. Players are ranked and ordered (in the scoreboard) based on this information. However the King will always remain on top regardless of score.
11- Number of Frags (Kills) each player has.
12- Connection information, this shows a player's ping and connection time.

8. File Formats Used

***.UT2MOD:** This file format is used as a mini-installer for the Checkmate modification. It contains the compiled checkmate mod in its entirety, including textures[2], maps, and models (if applicable)

***.U:** This file contains the compiled source, and may also contain textures, player models, and other media

***.UC:** Source files (text format), to be edited using WotGreal

***.INI:** This file format is used for initialization settings. Checkmate.ini will contain all game-related data that is required to be set upon starting UT2003. These INI files are also used when compiling code and for creating UT2MOD files

***.INT:** Contains localization information (used by UT2003 for international language settings)

***.UT2:** This is the file format used for map files. While creating maps is not part of our requirements per se, the game will need to use the existing UT2003 maps in order to run.

***.UPL:** Used to define player model attributes. It maps textures to a player model

***.DDS:** Direct Draw Surface; the file format used to store textures. It is a very efficient scheme, incorporating lossless compression techniques.

9. Algorithms

PlayerKilled

Every time a player frags another player, they are awarded points based on their current class and the victim's class. If they frag themselves or a teammate (if friendly fire is turned on), they lose a point. Players are not awarded points for fragging a player of a lesser class (i.e. a Queen fragging a Pawn)..

```
Function playerKilled(killer, victim):
        if (killer.team == victim.team):
                killer.updateScore(-1)
                return
        killerPnt = killer.pclass.pntValue
        victimPnt = victim.pclass.pntValue
        if (killerPnt <= victimPnt):
                killer.updateScore( ceil(victimPnt/killerPnt) )
```

Example: *Trinity* (Pawn) frags *Cypher* (Pawn)

killer = Trinity, victim = Cypher
Trinity.team = black, Cypher.team = white
Trinity.pclass = pawn, Cypher.pclass = pawn

1. Trinity is not on the same team as Cypher
2. killerPnt = 1, victimPnt = 1
3. Trinity's point value is equal to Cypher's point value, so Trinity is awarded one (*ceiling*(1/1) = 1) point

[2] A texture is an image that can be mapped to a three-dimensional object or onto a flat surface.

KingKilled
When the King is fragged the score of the winning team is updated (+1) and a new round is initiated. Before the new round starts a King is chosen for each team (based on the highest scoring player of each team. Then all the players' credits are reset to 0 and are respawned into the game world.

```
Function KingKilled(killer, victim):
        playerKilled (killer, victim)
        victim.opposingTeam.score++
        createNewRound()

Function createNewRound():
        for each team t
                leader = t.getLeader()
                leader.nextClass = king

        for each player p
                p.credits = 0
                p.respawn()
```

Example: *Morpheus* (White King) is fragged by *Trinity* (Black Knight)
killer = Trinity, Victim = Morpheus
Trinity.team = black, Morpheus.team = white
Trinity.pclass = knight, Morpheus.pclass = king

1. Trinity is awarded four points $\left(ceiling\left(\frac{10}{3}\right)\right)$.
2. Morpheus's opposing team (Black) is awarded one point.
3. New leaders are found.
4. New leaders are set.
5. All players' credits are set to 0.
6. All players are respawned into the world

KingReplacement
When a player, who is controlling the King leaves the team (by disconnecting or switching teams), they lose control of the King. One way to handle this would be to consider it a forfeit and award the opposing team. The problem with this is that the entire team is punished for the legitimate actions of another player. A more reasonable alternative is to find a player to take over as King. The strategy here is to find the highest scoring player and have them possess the King. If a replacement is not found (no players left on the team) then it is considered a forfeit. The opposing team would score a point and a new round would begin. In this case however, the team is empty so no one is really punished.

Note: Possess refers to a new player taking over the *entity* of the King. This means keeping the King's stats (health and position).

```
Function KingReplacement(oldKing):
        newKing = getLeader()
        if (newKing == none)
                kingKilled(oldKing, oldKing)
        else
                newKing.UnPossessed()
                newKing.Possess(oldKing)
```

Example: *Morpheus* (White King) leaves the server before end of round
oldKing = Morpheus

1. Set the new King to the player with the highest score.
2. Switch is the only other player on the White team so she is set to be King
3. Switch is respawned into the world as the King (keeping all the old stats. ie: amount of health)

LaserGuidedMissle

The Queen's rocket launcher will have a mode of fire whereby the weapon paints a laser dot that can be used to guide the direction of the rockets. When the user activates the laser dot, they can move it around to redirect the rockets. The position of the laser dot is determined by a straight vertex from the player to the first solid object (i.e. a wall or another player) that player is currently targeting. So the location of the laser do is constantly changing.

If we ignore the client-server model, the solution is somewhat simple: add a velocity modifier to the rocket towards the location of the laser dot. More care must be taken when we factor in the client-server model.

Since the server can only update the rocket's true location in an unpredictable manner, each client must be able to predict the path of the rocket. The psuedocode below would be called for every game tick[3] for each rocket if the targeting laser exists. This is executed differently depending on whether we are running on the server or running on the client. The server essentially updates the location of the rocket and applies the new velocity. The true location (ServerLocation) is sent to the clients at a regular but unpredictable interval. When a client receives the location update, it is applied to the local simulation. The client also predicts the new velocity. So the client duplicates the work that the server does to simulate the path of the rocket.

```
if (TargetLaserDot exists):
        if (ROLE is server):
                ServerLocation = Location
        else if ( ServerLocation != vect(0,0,0) ):
                SetLocation(ServerLocation)
                ServerLocation = vect(0,0,0)

        NewRotation = rotator(TargetLaserDot.Location - Location);
        SetRotation(Rotation + Normalize(NewRotation - Rotation)*DeltaTime*3)
        Velocity = Speed * vector(Rotation);
```

Example:

A Queen player activates the laser dot and fires straight ahead.

1. Client iteration:

1.1. The role of process is not the server, and we haven't received any updates, so the true location doesn't get updated.

1.2. We calculate NewRotation which represents the rotation from the missile to where the laser dot is. In this case the missile is already on that path, so NewRotation is the same as Rotation.

1.3. We update the missiles rotation to be a time-weighted average between NewRotation and Rotation, once again this has no effect since the missile's path.

The missile continues on its path, then the player changes the position of the laser dot and points it to the ceiling (straight up).

2. Server iteration:

[3] A tick is a function called by the Unreal engine for every frame of animation.

2.1. The role of the process is the server, so we update ServerLocation; this value is automatically updated to the clients.

2.2. NewRotation is calculated as the rotation from the missile's position to the location of the laser dot.

2.3. Now we update the rotation of the missile to turn towards the laser dot.

3. Client iteration:

3.1. The role of the process is not the server, but the ServerLocation has been updated, so we update our local copy of the missile's location

3.2. Calculate NewRotation

3.3. We continue with the prediction code by updating the local copy of the rotator.

Appendix A: Equations

Equation 1: Point system formula

if (EnemyClass ≥ PlayerClass)

PointsAwarded = $ceiling\left[\frac{EnemyClass}{PlayerClass}\right]$

else

PointsAwarded = 0

Appendix B: Player Classes

Pawn

Point value: 1
Credits required to obtain: 0
Weapon(s): Assault Rifle
Ability: Kamikaze
Skill(s): <none>
Drawback: very Low HP, limited weapon arsenal
Description: The Pawn will be the most widely used Player Class as he/she is the starting character, cheapest (points wise) character and has the highest reward for kills. The Pawn is the weakest Player Class.

Knight

Point value: 3
Credits required to obtain: 3
Weapon(s): Grenade Launcher, Assault Rifle
Ability: <none>
Skill(s): Higher jumping, Reduced splash damage
Drawback: Limited attack range
Description: The Knight will be a very acrobatic, offensive class.

Bishop

Point value: 3
Credits required to obtain: 3
Weapon(s): Sniper Rifle, Assault Rifle
Ability: Cloaking
Skill(s): Faster running
Drawback: Low HP, weak close-combat ability
Description: The Bishop is the stealthy, sniper class. He will be greatly disadvantaged in short range battle.

Rook

Point value: 5
Credits required to obtain: 5
Weapon(s): Chaingun, Assault Rifle
Ability: Turret mode
Skill(s): <none>
Drawback: Slow running, Reduced jump, No double jump
Description: The Rook can absorb a large amount of damage. He is very slow but has strong offensive & defensive capabilities

Queen

Point value: 9
Credits required to obtain: 9
Weapon(s): Rocket Launcher (laser-guided missiles), Assault Rifle
Ability: Off-hand hook
Skill(s): Slightly faster running
Drawback: <none>
Description: Queen will be the most powerful offensive character. She will deal a lot of damage and get many kills, but will not earn very many credits, due to the ratio scoring system (see **Equation 1**)

King

Point value: 10
Credits required to obtain: (Assigned, not applicable)
Weapon(s): Gauntlets
Ability: Recall
Skill(s): Faster running, King's Aura
Drawback: Limited range of attack
Description: The King is a melee fighter with command abilities. He is the focal point of the team, as the game revolves around his ability to out-last the opposing team's King.

Appendix C: Algebra in Unreal

1. **Actor locations:** every actor (an object that exists in a level) has a built-in location ("vector Location") that represents its position in the level. The location is represented by a three-dimensional vector that originates at the centre of the level that extends to the actor.
2. **Rotators:** an actor also has a rotator ("rotator Rotation") that represents the direction they are facing. A rotator has three angular components: pitch (up and down, like nodding your head "yes"), yaw (left and right, like shaking your head "no"), and roll (like tilting your head to the side). If a particular actor has a velocity, updating its rotator will also change its trajectory. A rotator can be derived from a vector, as seen in the pseudocode for the laser guided missile.