Heaven & Earth, Inc.

presents

Tale of Time

Steven Halim
Jefry Tedjakusumo
Lucas Agussurja
Fabianus Mulawadi

November 3, 2004

Contents

Chapter 1

Functional Document

1.1 Game Mechanics

1.1.1 Core Game Play

- You are Mark, a treasure hunter in search of a legendary treasure.
- Explore a whole island for the treasure, equipped only with a shotgun.
- Locate an ancient temple within the island.
- Solve the puzzle given to enter the temple.
- Locate an ancient artifact in the temple, fighting monsters along the way. The artifact is supposed to help you kill the final boss.
- Confront the final boss and claim the ultimate treasure.

1.1.2 Core Game Flow

You're playing as Mark, a treasure hunter in search of a legendary treasure. You're to explore a mysterious island, where the ultimate treasure is. This treasure is guided by a powerful guardian. With your simple gun, you can hardly do any damage to this guardian.

You may want to first locate the ancient artifact that will bestow upon you the special ability to inflict damage on the guardian. This ancient artifact is located inside an ancient temple, which in turn is guided by weaker monsters, that you can handle with your simple gun.

The artifact will help you to slay the guardian. You may try to steal the treasure without confronting the guardian, but it will be impossible since the treasure is protected by a barrier that will vanish only after the guardian is slain.

1.1.3 Characters/Units

- Mark: The treasure hunter you are playing as. His basic actions are walking, jumping, and shooting. Mark is equipped with one gun only throughout the game. With the gun, he is supposedly unable to defeat

Minotaur, but after gaining a power-up from a particular artifact, he will be able to defeat Minotaur.

- Minotaur: The final boss who is guarding the ultimate treasure. He has a regeneration power that makes him almost immune to normal attacks from Mark's gun.
- Critter: There are many small monsters lurking inside the temple that will attack Mark if he comes closer to them. Mark can either defend himself by shooting the monsters or flee to avoid fights.

1.1.4 Game Play Elements

- Normal Gun: The gun Mark carried with him from the beginning of game.
- Super Gun: Mark's gun that has been transformed after Mark acquired Tiger Balm. It has the capability of inflicting damage to Minotaur.
- Tiger Balm: An artifact from ancient civilization that is said to be able to grant great power to its owner.
- Talking Cock: The ultimate treasure which is said to be capable of reversing the flow of time.
- Pulau Ubin: A mysterious island in which the legendary treasure, the Talking Cock, is said to exist.
- Ancient Temple: A temple in the island that houses the Tiger Balm.

1.1.5 Game Physics and Statistics

There are two settings in this game: outdoor setting and indoor setting.

- Outdoor Setting
  The island is to be surrounded by water, and populated with: trees, a temple and the guardian. In navigating Mark should not be able to go into the water, run through the tress, or the temple or the guardian. Mark should be able to jump, following the law of gravity. Mark can look up and down to a certain angle, and move up and down according to the height of the terrain. Mark should be able to shoot the guardian, and the collision detection should be as precise as possible.

  If the guardian is behind a tree, the bullet should hit the tree and not the guardian. In return, the guardian is able to react to attack. The guardian has to be able to shoot back at Mark, and Mark can see it coming, giving him a chance to avoid the attack. If Mark is too close to the guardian, the guardian can deliver a blow that would reduce Mark's life to zero. The damage inflicted by the shots must take into consideration the distance of hit: shots from close distance should inflict more damage than ones from farther.
- Indoor Setting
  The inside of the temple is a labyrinth populated with monsters. In navigating, Mark should not be able to run through walls. Same as the case

with the guardian, Mark and the monsters should be able to shoot at each other. Their shots must not go through walls and the collision detection should be as precise as possible.

For the monster in the temple, however, only certain part of the monster (the eye) that can take in damage, hence to kill the monster, Mark must shoot at its eye. The damage must take into account the distance of hit. The inside of the temple is supposed to be dark, and this limits Mark's vision, but the monsters do not have this limitation, since that's what the big eyes are for.

1.1.6 Artificial Intelligence

- Minotaur AI:
  - Minotaur always walks around the treasure in normal situation.
  - If he detects Mark's presence, he'll prepare to fight.
  - Then, he'll try to close the distance between him and Mark. He'll also try to always turn his body to face Mark, within a certain degree.
  - After he gets close enough to Mark, he'll start attacking using fireballs. If the range is very close, Minotaur will unleash its lethal attack upon Mark.
  - If Mark retreats far enough, Minotaur will switch to normal mode and go back to guard the treasure again.
  - Occasionally, while Minotauris patrolling, he will take a look at the Talking Cock. If he sees Mark, he'll switch to attack mode.
- Critter AI:
  - Will attack if Mark's within their viewing area using a long range attack.
  - No blocking/dodging capability.
  - Capable of giving a limited chase, as long as Mark is within its viewing distance.

1.2 User Interface

1.2.1 Flowchart

(See figure 1.1).

1.2.2 Functional Requirements

Here we explain the different types of screens mentioned in figure 1.1.

- Menu: In a menu screen, player is presented with GUI components for him to make choice of actions to be taken. He can also customize the game; for example, setting difficulty level, setting god mode, etc.

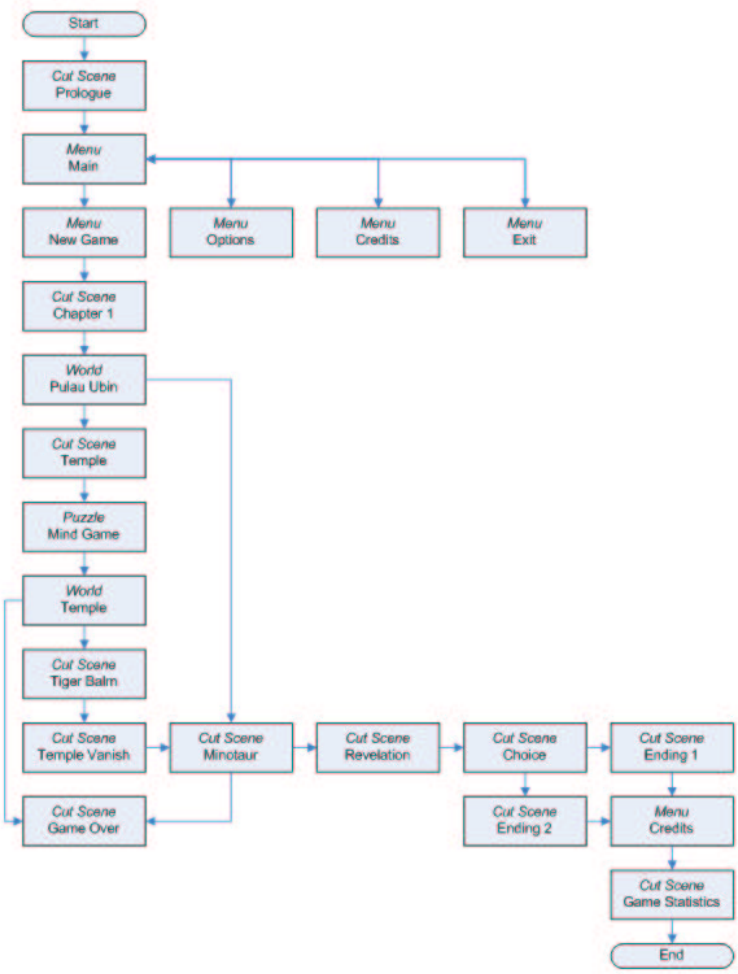

Figure 1.1: Possible Transitions Between Screens

- World: A world screen is where the player actually plays the game. There will be a character for the player to order (e.g. to move around, to shoot, etc).
- Cut scene: To explain/tell part of stories to the player, we make use of cut scenes. In this type of screen, there is illustration with scrolling text telling the story. We may use this screen to ask the player for input, too.
- Puzzle: In a part of the game, there is a situation where the player is required to solve a problem before he/she can proceed. We use this screen to serve such situation.

1.2.3 Mockups

For rough guide to designing the screens, please refer to figure 1.2 and figure 1.3.

Figure 1.2: Model cut scene

Figure 1.3: Model island

1.3 Art and Video

1.3.1 Overall Goals

The goal is to give off a feeling of mystery and exploration, so we must make the player feel that he really is stranded in a mysterious island unknown to him.

1.3.2 2D Art & Animation

GUI

- Command button: used in menus for player to indicate his/her choice.

Terrain

- Trees: to be placed randomly throughout the island.
- Sky: dark blue with a bit of grey, to give a shadowy nuance.

- Landscape: grass mixed up with tundra, since the island is located near the north pole.
- Temple Wall: white brick.
- Temple Ceiling: limestone.
- Temple Floor: brown granite.

Game Play Elements

- Tiger Balm: texture for Tiger Balm object.

Special Effects

- Gunfire: shown every time the gun is used.
- Minotaur Smoke: smoke effect that keeps surrounding the guardian.
- Talking Cock Barrier: sparkle/glimmer effect to indicate there's barrier around the treasure.

1.3.3 3D Art & Animation

Models

- Mark: just a gun and a hand holding it.
- Minotaur: a monster that looks tough and menacing.
- Ancient Temple: a mysterious temple that looks old/from ancient civilization.

Textures

- Minotaur: a monster that looks tough and menacing.
- Ancient Temple: a mysterious temple that looks old/from ancient civilization.

Animations

- Mark shooting: The gun recoils and Mark gets shaken a bit.
- Gun reloading: Mark puts in new ammo into his gun.
- Minotaur normal attack: He makes a throwing movement toward Mark.
- Minotaur fatal attack: He leaps toward Mark.

Special Effects

- Minotaur normal attack: fireball from his hands.

1.3.4 Cinematics

All scenes in this game consist of scrolling text story and voice narrative combined with still images as visual helpers.

- Opening scene
  This scene tells the player the background story–the beginning and the reason of his journey.
- Guardian scene
  This scene appears when he gets close to the guardian. It tells him also that the treasure can't be taken unless the guardian is defeated.
- Temple scene
  This scene appears when he is about to enter the temple. He will be asked to solve a simple puzzle before entering.
- Tiger Balm scene
  This scene appears when the player has managed to get the Tiger Balm–a power-up item.
- Temple Vanish scene
  This scene appears when the player leaves the temple. It tells him that the temple disappears mysteriously into thin air.
- Ending scene
  This scene tells the player the conclusion of his journey. The content depends on the choices he made when playing; it could be the first ending, or the second one.

1.4 Sound and Music

1.4.1 Overall Goals

The goal is to give off a feeling of mystery and exploration. The player must feel that he really is stranded in a mysterious island unknown to him.

1.4.2 Sound Effects

Units/Characters

- Mark scream when attacked
- Monster shriek when shot

Terrain

- Monster howl: occasionally heard in outdoor.

Special Effects

- Normal Gun: sound of a shotgun.

1.4.3 Music

Shell Screen

- Title screen: slow, happy tune.
- Credits: slow, happy tune.

Level Theme

- Outdoor: a slow music, giving away a sense of loneliness and long journey.
- Indoor: a slow music, giving away an eerie/mysterious air.

Situations

- Outdoor combat: a fast-paced music.

1.5 Story

1.5.1 Prologue

Mark was an archeologist and treasure hunter. Together with his girlfriend Ivy, he traveled from one place to another in search of remains of ancient civilizations and their treasures.

However, one day, they got chased by vicious tribes. They managed to make a run of it, but Ivy was mortally wounded, and she died shortly after. Mark was totally broken down, and he stopped traveling.

Mark would spend every day just spacing out and mourning over Ivy, until one day he heard about a treasure named Talking Cock, which was said to be able to reverse time. Rumors said it was located in Pulau Ubin. Determined to return Ivy back to life, Mark went for his final journey.

1.5.2 Act 1 - Pulau Ubin

Armed only with his trusty revolver, Mark went on search for Pulau Ubin. It didn't take him long, however, to locate the island, given his skills and determination. Soon after he arrived in the island, Mark realized he wasn't alone. There were many vicious monsters lurking within it, making Mark simply unable to let down his guard.

After wandering around for several days, Mark discovered a cave. Curious, he entered to check. To his surprise, the very treasure he was looking for was there. It was, however, at all time guarded jealously by a Minotaur.

1.5.3 Act 2 - Failed Attempt

Mark, being determined to get the treasure on his hands no matter what, confronted the Minotaur in a battle. The Minotaur, however, was impervious to all his attacks; not a single bullet put a scratch on his body. Realizing he was at a disadvantage, Mark fled, and luckily, the Minotaur didn't go after him.

Afterwards, Mark decided to explore the island further, hoping to find clue on how to defeat the Minotaur. After a while, he came across a curiously-looking ancient temple.

1.5.4 Act 3 - Ancient Temple

Mark decided to inspect the ancient temple and entered. The inside was dark; luckily for Mark, there's a kind of fluorescent moss that gave off enough light for him to find his way around. Unluckily, the temple was infested with monsters too. Mark was attacked and forced to defend himself.

After the monsters stopped attacking, Mark looked around and realized that he had entered too deep into the inside of the temple. To make things worse, the inside was actually a labyrinth with a lot of similar paths. Thus, Mark was trapped, unable to find the way out.

On the fifth day, just when he was about to give up hope, Mark saw a glimmering object from afar. He approached it; it was a very big jewel, put on a pedestal.

Never in his life had Mark seen one so brilliant and magnificent. The surrounding areas were brightly lit by the jewel's brilliance. Mark read an inscription written on the pedestal. It was written:

TIGERBALM

From the inscription, Mark knew that the jewel was called Tiger Balm, and that it fell from the sky long time ago. It was also said that whoever possesses it will be granted divine power from the heaven.

Mark took the jewel and held it in his hands. The jewel felt warm, and he felt a surging power flowing from the jewel to his body. Mark was energized and he felt much stronger than before.

1.5.5 Act 4 - Destroy Minotaur

Having acquired new strength, Mark gained confidence and fought the Minotaur once again. He really had become stronger; this time he could actually damage the Minotaur. The fight was intense and fierce, but in the end the Minotaur was proved to be no match for Mark. Mark was victorious.

To Mark's surprise, the Minotaur turned into human, and he spoke to Mark. Long time ago, Minotaur was a human; his name was Ah Beng. He found the Talking Cock, and used it to reverse time. Time was indeed reversed; however, the Talking Cock was cursed, and anyone who uses it will be destined to guard it until someone else gets it. Ah Beng was no exception; he was turned into Minotaur and had been guarding the Talking Cock for hundreds of years.

Thus, Mark was faced with a dilemma. He could use it to reverse time and Ivy would come back to life. However, he would turn into Minotaur and would not be able to meet her forever. It's a difficult choices, but he had to decide.

1.5.6 Epilogue

After hesitating for a while, Mark decided not to use the Talking Cock. He realized that it's not for humans to go against fate, and started to accept the

fact that Ivy was no longer with him. He left the Pulau Ubin and went home, living a quiet life without any adventure.

1.5.7 Epilogue - Alternate Ending

After hesitating for a while, Mark decided to use the Talking Cock. Time was reversed back to the point where Ivy was still alive, while Mark changed into Minotaur. Ivy didn't remember anything about Mark, though, and lived a quiet life without any adventure.

1.6 Level Requirements

1.6.1 Level Diagram

(See figure 1.4).

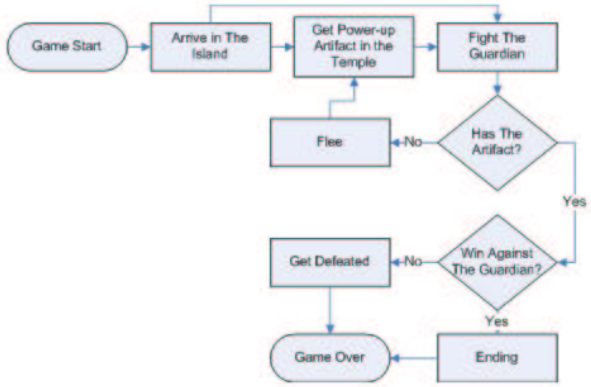

Figure 1.4: Mission Tree

1.6.2 Asset Revelation Schedule

- Gun: available all the time.
- Tiger Balm: acquired after finishing temple task.
- Super Gun: automatically upgraded upon acquiring Tiger Balm.
- Talking Cock: can be acquired upon defeating the guardian.

Chapter 2

Technical Document

2.1 Game Mechanics

2.1.1 Platform and OS

- Hardware Platform: Intel x86-compatible PC.
- Operating System: Windows XP.
- Hardware Requirements: 128MB Video RAM, 256MB RAM, 20MB HDD.

2.1.2 External Code

- Microsoft DirectX: Graphics API developed to work under Microsoft Windows.
- TV3D: Game engine developed by TrueVision3D, which makes use of Microsoft DirectX.

2.1.3 Code Objects

In figure 2.1 we show the class diagram for our game. Those are all the classes that we implement in this project. The two dynamic libraries which are provided by the engine and used in this project are:

- TV3DEngine Library: written using DirectX8.1 and provide functions for accomplishing 3D abstraction. It wraps some of the more powerful features of DirectX8.1 into readily used functions and procedures.
- TV3DMedia Library: written using DirectX8.1 DirectSound and provide functions for accomplishing media effects, such video and sound effects. It supports common sound typefiles, 3D Sound and Effects.

In the following, we give a brief description of the classes:

1. MainWindow: this is where the main loop of the game resides, it calls the MenuScreen and calls the corresponding world when the MenuScreen returns newgame, or exit the game when it returns exit.

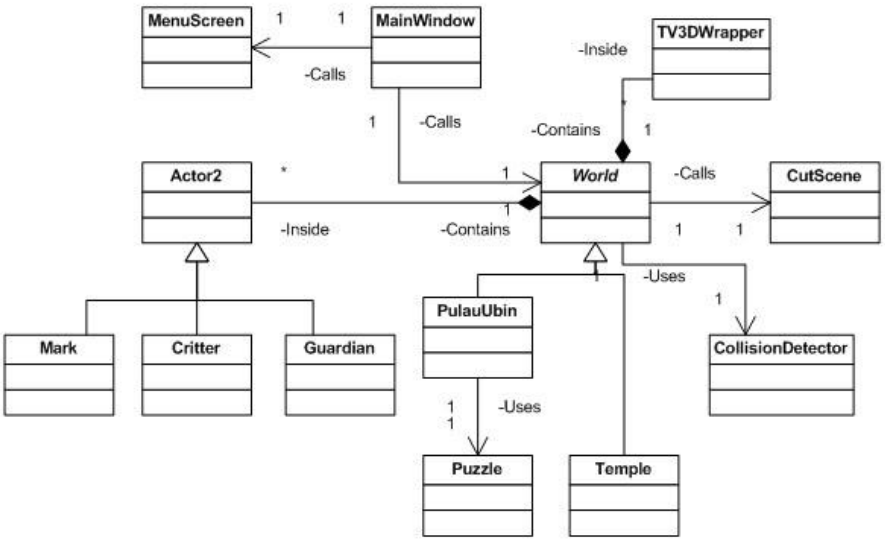

Figure 2.1: Class Diagram

2. MenuScrenn: this is the class responsible for creating menu screen and getting the user input at the start of the game.

3. World: this is the general class implementing the world. It has two child-classes: PulauUbin and Temple. It contains TV3DWrapper and Actor2 classes. It also have a reference to CutScene class and uses the Collision-Detector class.

4. PulauUbin: this is the class that implements the outdoor setting (the first level) of the game. It uses the class Puzzle.

5. Temple: this is the class that implements the indoor setting (the second level) of the game.

6. Puzzle: this is the class to hold all the puzzles that are used in the game. In our game, we have only one puzzle.

7. CutScene: this is the class to store all the cutscenes that will be needed during the game, and can be called by the corresponding world.

8. CollisionDectector: this is the class to implement the collision detection algorithm.

9. TV3DWrapper: this is the class that stores the common objects provided by the engines that are used by the different World classes.

10. Actor2: this is the class provided by TV3DEngine and not implemented by us. We use this class as the super class for our character and monsters classes: Mark, Critter, and Guardian.

11. Mark: this is the class that implements and stores all the information about the player.

12. Critter: this is the class that implements the monsters inside the temple (the indoor setting).

13. Guardian: this is the class that implements and stores the information about the guardian (the final boss).

We have tried our best to apply the concepts from Software Engineering to this project, such as: Object-Oriented (abstraction, generalization, information-hiding), modularity, and scalability(extensibility).

2.1.4 Control Loop

Refer to the figure on the next page. In the figure, we show the overview of the running of the game.

2.1.5 Game Object Data

Here we'll describe our characters and worlds' representation, and any data they contain.

- PulauUbin
  The PulauUbin object has the following variables inside (simplified):
  - TVMesh temple; storing the object temple.
  - D3DVECTOR treePositions; for storing the poition of the trees.
  - float renderDistance; for specifying the minimum distance of objects in this world to the player, such that the objects are redered.
  - TVParticleSystem shootParticle; to create the special effect of the guardian being hit.
  - TVLightEngine LightEngine; to provide the world with lighting information (only apply to temple).
  - TVPath wayMarkers; this is use to navigate the guardian in this world.
  - float waterLevel; to specify the height of the water.
- Guardian
  The Guardian object has the following variables inside (simplified):
  - float HP; guardian's hit points
  - D3DVECTOR direction; vector indicating the guardian's direction.
  - D3DVECTOR destination; indicating the next destination of the guardian.
  - D3DVECTOR currLookat; indicating the guardian's current lookat position.
  - D3DVECTOR nextLookat; indicationg the guardian's next lookat position. Both currLookat and nextLookat are used in interpolating the guardian when turning.
  - D3DVECTOR attackTo; vector indicating the direction of the attack of the guardian.

  - bool turning; indicating whether the guardian is turning.
  - float walkingspeed; indicating the speed of the guardian while walking.
  - TVParticleSystem guardianEnergy; to create the special effect that covers the guardian with smoke.
  - D3DVECTOR[] center; for storing the center of the bounding spheres.
  - float[] radius; for storing the radius of the bounding spheres of the guardian.

- Player
  The player object has the following variables inside (simplified):
  - D3DVECTOR position; vector indicating player's position in 3-D world.
  - D3DVECTOR direction; vector showing player's direction.
  - float hp; player's hit points.
  - float walk; player's walking speed.
  - float strafe; player's strafing speed.
  - bool shoot; indicating if the player's shooting.
  - bool powerUp; indicating if the player's got a power-up item.
  - bool crouch; indicating if the player's crouching.
  - bool jump; indicating if the player's jumping.
  - int ammo; number of ammo the player currently has.
  - string objective; the current objective the player has to achieve.

- Critter
  The critter object has the following variables inside (simplified):
  - float hp; critter's hit points.
  - bool isDead; indicating if the critter's already dead.
  - float atkSpeed; indicating the speed of critter's attack particles.
  - D3DVECTOR atkTarget; vector indicating to which the attack should be aimed.
  - Random random; pseudo-random generator to create 'spray' effect for the attack.

- Temple
  The temple object has the following variables inside (simplified):
  - List critters; container for all critter objects inside the temple.
  - Mesh mesh; mesh for the temple/maze's structure–the ceiling, floor, and walls.
  - templeDim; relative dimension of the temple as stated in temple map.
  - artPos; relative position of the artifact–Tiger Balm.
  - exitPos; relative position of the exit door.
  - pathMap[]; 2-D array storing location of walls.
  - float blkSize; indicating size of one block of temple map in pixels (see map below).

2.1.6 Game Physics and Statistics

In outdoor setting, we have a mechanism to detect collision between player and trees (static detection). We also have another mechanism to detect whether the guardian gets shot.

- Player bullet vs Guardian
  For the dynamic collision detection of mark shooting the guardian, we use the ray-sphere intersection algorithm to detect the collision.

  First, we select certain bones from the model, which approximate the whole model. For our guardian model, we have a total of 63 bones, out of which we choose 24. At the center of each of the 24 bones, we draw a sphere of different sizes for different bones. The result of this, we have 24 spheres that approximately cover the body of the model. We consider this good approximation, since we manage to cover almost all of the guardian body without overly covers unnecessary parts.

  When Mark is taking a shoot, a line is logically drawn describing the path of the bullet. This line will be tested first at a bigger bounding sphere covering the whole body of the guardian. If the line passes this test, it will then be tested at each of the smaller 24 spheres. Since these spheres have very small overlapping regions, we do not bother checking from the nearest to the furthest.

  Below is the algorithm used, described in mathematical form.

  Consider two points defining a line: $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$, then a point $P(x, y, z)$ that lies on the line can be described by the following:

$$\begin{aligned} x &= x_1 + u(x_2 - x_1) \\ y &= y_1 + u(y_2 - y_1) \\ z &= z_1 + u(z_2 - z_1) \end{aligned}$$

  Now, consider a sphere centered at $P_3(x_3, y_3, z_3)$ with radius r is described by:

$$(x - x_3)^2 + (y - y_3)^2 + (z - z_3)^2 = r^2$$

  If we substitute this equation to the line equation, we get a quadratic equation of the form:

$$au^2 + bu + c = 0$$

  where:

$$\begin{aligned} a &= (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 \\ b &= 2[(x_2 - x_1)(x_1 - x_3) + (y_2 - y_1)(y_1 - y_3) + (z_2 - z_1)(z_1 - z_3)] \\ c &= x_3^2 + y_3^2 + z_3^2 + x_1^2 + y_1^2 + z_1^2 - 2[x_3x_1 + y_3y_1 + z_3z_1] - r^2 \end{aligned}$$

  The behavior of the line over the sphere is determined by the value of $D = b^2 - 4ac$:

  - If $D < 0$ then the line does not intersect the sphere.
  - If $D = 0$ then the line is a tangent to the sphere.

  - If $D > 0$ then the line intersects the sphere at two points. The points can be calculated using the formula for solving the quadratic equation:

$$u_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  There are several considerations in using line-sphere intersection for dynamic collision detection:

  - It is simpler and easier to implement compared to line-box intersection.
  - The amount of data that we need to store and update is minimal, i.e. we need only the two points defining the line, the center point, and the radius of the sphere.
  - Our model allows us to use just several bounding spheres to get a good approximation.

- Player vs Trees
  The static collision detection between mark and the trees is done using simple distance check between mark's position and the trees' positions. Given mark's position and his direction at any point of time, and given the trees' position, we can calculate whether mark is in certain small distance to any particular tree, and whether mark is heading towards that tree. If the result of this test is true, then we conclude that a collision is detected and mark is not allowed to move. We do this checking exhaustively considering the amount of trees is not too many.

Inside the temple, we have also both static and dynamic collision detection mechanisms.

- Player vs Wall
  Since the maze inside the temple is drawn in blocks (see section 2.1.5), we know precisely where the wall begins and ends. Thus, the detection is quite simple. Every time player's position is updated, the mechanism checks for invalid move.

  For example, when the player is trying to move to the left, we check for the presence of wall to his left. If there is no wall, then we're done. But if there is wall, we check his distance to the wall. If the distance is within some threshold, we cancel his left movement only, so he will still be able to move forward.

- Player vs Critter
  The detection mechanism for this kind of collision is done in a similar way as what have been mentioned in section above 'Player vs Trees'. But, in this case the obstacle is the critter.

- Player bullet vs Wall
  To detect whether the bullet hits the wall, we utilize a simple 2D line-rectangle intersection algorithm. As what have been mentioned in section 2.1.5, the maze structure is in fact divided into blocks. And since the wall is connected to ceiling and floor at all times, we only need to perform a 2D operation instead of 3D operation.

When the player shoots, we check his shooting direction and check all the blocks within the area covered by his bullet. If the bullet hits the wall, our method returns 2D distance between player and the wall hit, otherwise it returns a very big number.

This is what we more precisely do:

- Get player's current position
- Get the bullet's target coordinate using formula:

$$\begin{aligned} x\_target &= bullet\_range \times \cos\alpha \\ z\_target &= bullet\_range \times \sin\alpha \end{aligned}$$

  where α is the angle of player's direction.
- From the two points, we calculate the equation for the bullet's trajectory; we get the line equation $z = mx + c$ where m is the gradient and c some constant.
- For every block that might be passed by the bullet, we perform line-rectangle intersection check. Here we treat each rectangle as four lines, and check our line against the four lines.
- If our line intersects any of the four lines, we check if the intersection point is between two parallel lines perpendicular to the one we intersect.
- If the point is indeed between the two, then we can conclude that the bullet hits the wall.

- Player bullet vs Critter
  The detection mechanism for this kind of collision is done in a similar way as what have been mentioned in section above 'Player bullet vs Guardian'. But, in this case we only need to specify one sphere for detection, as the critter is in the form of a big eye with legs. And, it's been decided that the only way to inflict damage to the critter is by shooting the eye.

  To make sure that the player cannot shoot critter behind the wall, when he shoots we check for collision with wall first, then check for collision with the critter. If the distance with the wall is less, it means the bullet is obstructed by wall, and we don't count the shoot as a hit.

- Critter attack vs Player
  The critter attack is in the form of finite number of particles 'sprayed' towards the player. Every time we are about to redraw the particles, we check the position of each particle and compare it to player's position. If the distance is close enough, it's counted as a hit, otherwise a miss. Since the computation may be quite expensive, only a limited number of particles is 'sprayed' in every attack.

2.1.7 Artificial Intelligence

Since we have two types of monsters, we have two different AI's. The guardian AI is obviously much more complex than that of critters. Here we explain both AI's:

- Guardian AI
  Before going into the AI of the guardian, we describe two things: navigation and viewing angle of the guardian that helps the AI of the guardian to work.

  To navigate the guardian over the terrain, we use waymarkers, these are points on the terrain specifying where the guardian can move to. So the guardian can only move from one waymarker to another waymarker. The locations of the waymarkers are placed near the treasure. When the guardian needs to turn, we use a linear interpolation to smoothen the turning.

  The guardian also has a limited viewing angle, i.e. from the current lookat direction, 45 degree to left and to the right. If the player is inside this region and at certain distance to the guardian, then the guardian can notice the presence of the player, otherwise no. This is implemented by drawing a line from the player to the guardian and then measure the angle between this line and the current lookat direction of the guardian. If this angle is between -45 to 45 dengree, and if the player is within some distance to the guardian, then the guardian can see the player.

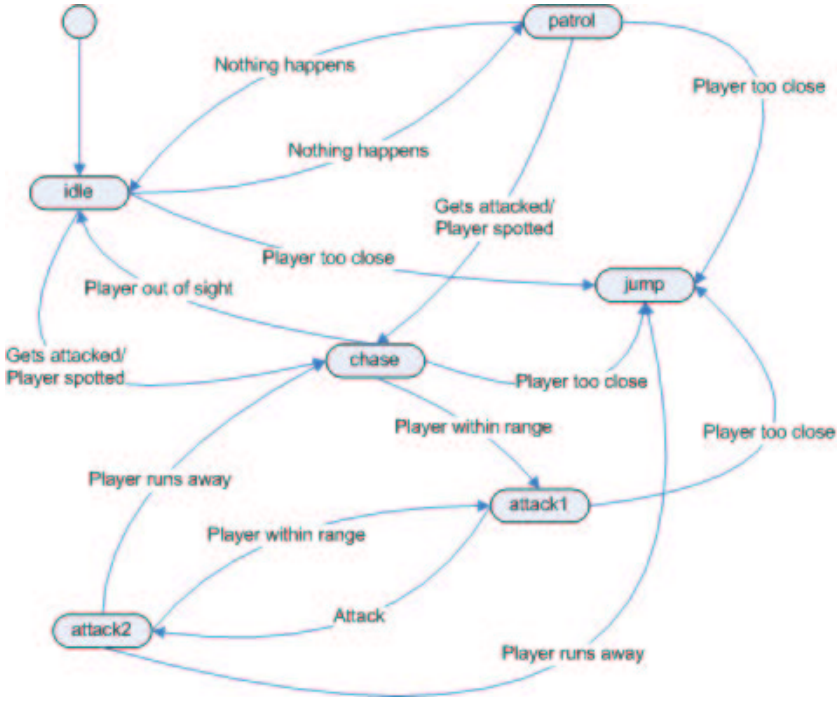

Figure 2.2: The Guardian's State Diagram

Figure 2.2 shows the behaviour of the guardian in the form of a state transition diagram. At any point of time, the guardian will be in one of the states. The actions that the guardian takes depend on which state the guardian is in. Transitions between states can happen because certain things change. In the following we enumerate all the states in greater detail:

1. State: Idle
   - Action taken: do nothing. When the guardian is in idle state, it will be facing the treasure.
   - Possible transitions:
     * Some time has elapsed and nothing happens: change to patrol state.
     * Player spotted: change to chase state.
     * Attacked by the player: change to chase state.
     * Player spotted and the player is too close to the guardian: change to jump state.
2. State: Patrol
   - Action taken: walk to the next nearest waymarker.
   - Possible transitions:
     * Some time has elpsed and nothing happens: change to idle state.
     * Player spotted: change to chase state.
     * Attack by the player: change to chase state.
     * Player spotted and the player is too close to the guardian: change to jump state.
3. State: Jump
   - Action taken: jump towards the player and deliver a final blow to the player that reduce the life of the player to zero.
   - Possible transition: none, if this happens then it's game over.
4. State: Chase
   - Action taken: run towards the waymarker that is nearest to the player.
   - Possible transitions:
     * Player become out of sight: change to idle state.
     * Player become too close to the guardian: change to jump state.
     * Player is within the attack range: change to attack1 state.
5. State: Attack1
   - Action taken: throw a flame ball to the player.
   - Possible transitions:
     * Player become too close to the guardian: change to jump state.
     * The flame ball is thrown: change to attack2 state.
6. State: Attack2
   - Action taken: Action2 is the state where the guardian is waiting before throwing another attack to the player.
   - Possible transisitons:
     * Player runs away: change to chase state.

  - Some time has elapsed after the previous attack: change to attack1 state.
  - Player become too close to the guardian: change to jump state.

The AI is implemented by constantly checking and updating the state the guardian, and take the corresponding action.

- Critter AI
  Critter AI is basically consisting of detecting player and attacking, plus limited movement.

  In each loop, we compute each critter's distance from the player. If the distance is less than some threshold, we next check if there's actually wall blocking the view using method mentioned in section 2.1.6 item 'Player bullet vs Wall'.

  Also, as long as the player's within their field of view, the critter will actually try to approach the player. It will walk approaching the player in between its 'spray' attacks. If the player is gone from its range of view, the critter doesn't go back to its original position; instead, it stays where it currently is.

2.2 Art and Video

2.2.1 Graphics Engine

- Outdoor Settings
  The landscape for the outdoor setting is specified using height map. There are basically four main components in the outdoor setting: the trees, the temple, the guardian, and the sky.

  The trees in this setting are just billboards placed semi-randomly over the terrain. It is random because we do not manually specify all the positions of the trees, but use the random function to generate the positions. It is semi-random because we fix the seed that is used to generate the positions. All the trees are generated from just one image which is scaled randomly to give the effect that the trees are different from each other. The amount of trees should be around 500 to 1000. In order to reduce the rendering cost, not all the trees will be rendered, but only the trees that are reasonably visible to mark at any point of time. The only information that we need to store about the trees is their positions. These points are stored in an array of vectors, and is used for static collision detection between mark and the trees as described in the game physics and statistics section.

  The temple that we use is a 3-D polygon mesh in x file format. And because it is an x file format, it can be enhanced using lighting in the game. TV3D engine supports lighting for x file format but not mdl format.

  The guardian is a 3-D polygon mesh model in mdl format. It is a skeletal based model. We can draw information about each specific bone especially its position in the world. This information is used to create bounding spheres for the guardian for dynamic collision detection as described in

game physics section. We use the particle system provided by the TV3D engine to create a special effect on the guardian, where this guardian seems to be always emitting great heat that its body is always covered with smoke. We also use particle system to mark the point of hit in the guardian body when mark manages to shoot on target.

Both the temple and the guardian is not always rendered to the world, but only when they are in certain distance to mark. The outdoor setting is logically inside a big box, where we place images on the inner side of box to give the effect of the sky and distance view.

- Indoor Settings
  The inside of the temple is modeled using simple textured mesh. It is build based on predefined temple map. More precisely, five different meshes are used to build the maze; for the ceiling, floor, upper wall, lower wall, and exit door. We make use of one more mesh to model our artifact that's stored within the maze.

  The critter is a 3-D skeletal based model stored in mdl format. Its attack is simply a particle system.

  To increase performance, only walls and critters within some distance from the player is rendered.

2.3 Sound and Music

Sound and music are loaded during initialization of a world using facility from the game engine we're using.

2.4 Level Specific Code

For the indoor setting, when the temple is loaded, it reads from temple map file and build the maze based on the file content. Following is a sample of temple map:

```
------------------     Legend:
-----*            -         * critter
---------*------- -         - wall
---*         * --*-         T treasure
-*-- ---- ---- -- -         M player
- -- --*- -*T- -* -         X exit door
- -- -- - - -- -- -
-       -*   -    -
- -- --------- -- -
- --*-*   --*  -* -
- -- ----*----*-  -
-*--           ----
- -------*---------
-    *         M X-
------------------
```

This way, it's possible to create different maze easily if we so desire.

Appendix A

Development Strategy

The first thing to comment from this game project is that actually the total time spent for doing this project (total time from four of us summed together) is maybe 3 or 4 times higher than any previous course project that we have embarked so far (exception: Software Engineering project), but we conclude that the time spent are time well spent, since throughout this project, all of us gained our first ever experience in creating 3D games and managed to complete the whole game development process :). Almost in every single day since mid semester break; one of us is actually doing the coding while the others are busy with other modules. Therefore, our game "Tale of Time" is always progressing daily.

Regarding the team dynamics, we are lucky to have a team with uniform nationality and with uniform language (Indonesian), which is seldom encountered in a real life situation. We also located in the same lab in SoC (Computational Logistics Lab), we met almost every day (except weekends), thus communication is never been an issue for our team. Our team members also blessed with various background knowledge, which is useful for this game project. For example, almost all of us know how to use Adobe Photoshop, thus editing Artworks (say, to set transparent color) is simplified. We also can consider ourselves gamers, two of us actually beating DOOM 3 prior to this game project to get a look and feel of a state-of-the-art First Person Shooter game. Last but not least, coding also never been an issue for us, since we applied our software engineering skill for reading the not-so-well-documented TV3D API to master this 3D Game Engine quickly.

Next, we want to acknowledge that for this project, we didn't build any 3D model from scratch, but rather browse through the media folder in TV3D SDK, DirectX SDK, and Internet. The reasoning is simple: only two of us have experience with 3D modeling before (3D Studio Max) and their experience showed them that to build an aesthetically good 3D model, one needs to spend hours of labor, and we simply didn't have that time. We are lucky that TV3D provided one good MDL model for the monster, and then subsequent searching in the Internet lead us to a website which provide free Half-life models (MDL) [http://www.hlskins.free-online.co.uk/]. The time saved by not doing these 3D modeling is devoted for more improvement in other game development areas. For 2D artworks, it is a different case. We spend some time editing the textures, cut scenes etc to suit the look and feel of our game.

Finally, we acknowledge that we were a sort of over-promise when we propose our game concept to the lecturer. When we trace back those promises, we found out that we are unable to implement the following promises due to time constraint:

- Background music (MIDI): Initially we want to compose it by ourselves, but it turns out to be very difficult and at the end we use the Chrono Trigger Midi.
- Cave for the 3rd Level: Simply because we don't have time to add another level.
- Multiple endings: Previously we have several ideas but at the end we only have two distinct endings, whether you use Talking Cock or not, that's all.
- Two forms of the guardian (i.e. transforming into a stronger form after the first is defeated).

Appendix B

Division of Labor

- Fabi
  - Programmer: coding the whole temple labyrinth level from scratch (indoor environment setting), including the lighting, static/dynamic collision detection within (see sect. 2.1.6), and critters' AI.
  - Story writer
  - Documenter: writing almost half of documentation.
- Lucas
  - Programmer: implementing ray-sphere intersection checking for dynamic collision detection; coding guardian's bounding sphere and 'guardian vs bullet' dynamic collision mechanism; introducing trees into the island and related 'player vs tree' static collision detection; putting temple model into the outdoor setting, and setting the lighting.
  - Artwork artist: drawing some of artworks used in cut scenes.
  - Documenter: writing almost half of documentation.
- Jefry
  - Programmer: coding player object (including the movement and the gun); tidying up the code and abstracting different game objects into classes so that we can split our work and code each portion independently; handling animation coordination in our program; adding movement capability to critters inside temple.
  - Artwork artist: editing most of the artwork using Adobe Photoshop.
  - Other role: finding the necessary artworks from the Internet for our use, most notably the object for Talking Cock (the final treasure).
- Steven
  - Leader: organizing meetings.

- Programmer: writing the initial game code from scratch (very messy before transformed by Jefry); integrating all the codes; writing AI for the guardian; creating the interface for cut scenes; coding the menu screen; creating the mind game puzzle.
- Documenter: Writing errata for initial version of documentation; writing appendix content.
- Other role: Recording the voice over and converting it to mp3 format.

Appendix C

Problems and Solutions/Workarounds

C.1 Technical Problems

Here we present some problems that have more to do with technicalities and our workarounds to solve it.

- Experimentation problem
  It's very difficult to set some of the game's parameters, such as player's HP, player's attack power, and monster attack power.

  Solution: we play the game many times with different value to get the 'right' combinations. Also, we give the player an option of playing with different level of difficulty.

C.2 Integration Problems

Here we present some problems we encountered during integration of our codes, mainly when introducing world2 (Indoor - Temple) to world1 (Outdoor - Pulau Ubin).

- Program crashes during transition from world1 to world2.
  At first we thought this was because of incomplete disposal of resources such as textures/sounds. However, when we checked TV3D forum, we found that there're other people who experienced similar thing. Therefore, we conclude that this might be a bug related to TV3D game engine.

  Solution: instead of freeing resources, we try not to do so for resource type that seems problematic. What we do is that we reuse some of the resource previously loaded, such as texture. Also, it turned out that lighting in world2 wouldn't work after integration, so to achieve similar effect, we make use of fog effect inside the temple.

- Program crashes when player attempts to re-enter Temple.
  After the first problem is solved, there is a similar one occurring. After the

player entered the temple and went out, if he tries to re-enter the temple, the program will crash.

Solution: we make it part of the game that once the player enters the temple, he won't be able to go out until he accomplishes the objective in the temple. Then when he goes out of the temple, we show a cut scene saying that the temple has disappeared.

Appendix D

List of Artworks & Sounds

D.1 Artworks

- Minotaur: gs.mdl model from TV3D SDK. The 2D image of this monster is drawn by Lucas.
- Shotgun: v_shotgun.mdl model from TV3D SDK.
- One-eyed monster: taken from http://www.hlskins.free-online.co.uk website.
- 3D model for Temple (outside view): taken from the Internet
- Wall textures for temple maze: Edited from TV3D SDK artworks.
- Pictures for cut scenes: mainly taken from the Internet, edited to suit our color theme using Photoshop
- Atmosphere and cloud textures: taken from TV3D SDK.
- Tiger Balm: from the Internet.
- Talking Cock: from the Internet.
- Pendulum for opening scene: taken from some source on the Internet and edited.

D.2 Sounds

- Opening and menu screen MIDI: Chrono Trigger [Super Nintendo best game of 1995] theme song, since its theme revolves around Time too.
- Cut scenes voice over: recorded by ourselves; the voice actor is our housemate Andrew.
- Gunshot, monster shriek/howl, and player scream, other sounds from TV3D SDK or the Internet.