Tongji - Development

**Technical Design Document**

(Version 0.5)

Table of Contents

[1. Overview 3](#_Toc260387709)

[2. Requirements 3](#_Toc260387710)

[2.1. Brief 3](#_Toc260387711)

[2.2. Reference 3](#_Toc260387712)

[3. Dependencies 3](#_Toc260387713)

[4. Existing Technology 4](#_Toc260387714)

[4.1. Features 4](#_Toc260387715)

[4.2. Reference 4](#_Toc260387716)

[5. Implementation details 4](#_Toc260387717)

[6. Proof 5](#_Toc260387718)

[7. Issues 5](#_Toc260387719)

[8. Risks 6](#_Toc260387720)

[9. Estimates 6](#_Toc260387721)

# Overview

The overall architecture goal of the system is to make a funny and robust bomber man game.

**GDD**

**Game type:**

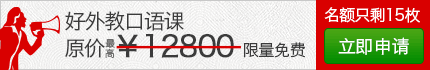
* Strategic, maze-based 3D game

**Game concept:**

* The game map is designed as a chessboard. The protagonist and enemies can only be moved step by step. Different obstacles are located in the map which will stop the characters.
* The protagonist can place a bomb in the chessboard without obstacles. Some obstacles and enemies around will be destroyed following the bomb blasts. The protagonist will also be killed if he/she is in the range of explosion.

**Game structure:**

* character



protagonist：the character who can be controlled by the player

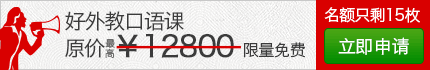
The protagonist can move to the every blank area and put a bomb in the blank area in the map.

enemy：the character which need to be destroyed

The enemy can move automatically to the every blank area in the map and can attack protagonist in the detected zone. You will lose when the protagonist meet the enemy otherwise you’ll win when all the enemies are killed.

If possible, the enemies can have different occupations, such as patrolman and Robocop. They have different duty, that is to say, they have different AI.

* bomb



The bomb can be placed in any blank area and will blast after a certain period. The explosion can destroy everything in the adjacent grid around except the rock. The power varies from different types of the bomb.

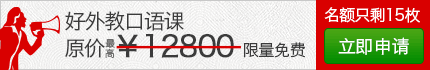
The explosion:

1. can kill the character

2. can destroy the wall

3. does nothing to the rocks

* map



The map is divided into grids. Every grid can be placed only one game element.

Blank area：the area where character can move and the bomb can be placed

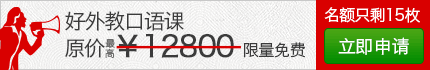
Rock：the obstacle which can stop the characters and can’t be destroyed

Wall：the obstacle which can stop the characters and can be destroyed by the bomb and the position it locates will turn into blank area.

Bonus: the bonus which can be picked up will add the number of bombs or can increase the speed, etc.

**Supported platform:**

* PC



# Requirements

## Brief

**Game Editor**

* Level data generator
* Level data exporter
* Visual map editor
* Basic console (undo/redo, erase, etc.)
* Alterable texture table
* Toolbar
* Preview tab (Alternative)

**AI system**

* Thinking
* Path finding

**Physical Engine**

* Particle simulation
* Collision detection

**3D Animation**

* character

**Render System**

* Light & Shading
* Shadow

**Sound editor**

* Sound effect
* Volume controller

**User interface**

* Menu Scene
* Options

## Reference

* <http://en.wikipedia.org/wiki/Game_Editor>
* <http://www.coronalabs.com/blog/>
* <http://gamedev.tutsplus.com/articles/workflow/make-your-life-easier-build-a-level-editor/>

# Dependencies

* Issues on XML Parsing
* Code balance between editor and game itself
* The character models
* Textures

# Existing Technology

## Features

**Game Editor**

**Level data generator & exporter**

* **Level data format**

We choose XML (or create our own format) as the very level data file (LDF) format. It is a textual data format with strong support via [Unicode](http://en.wikipedia.org/wiki/Unicode). The main purpose of LDF is saving positions of map elements (walls) and heroes, properties of game element (shape, solid/fragile, etc.)

* **LIBXML**

LIBXML is a 3rd party library used to create/parse XML files.

**Visual map editor**

* **Visual brush**

Users may draw walls and other map elements via brush directly on canvas.

* **Easy-to-use & Real-time feedback**

Scenes that rendered on the canvas should be exactly the same as what will be presented in the game, WYSIWYG.

**Basic console**

* **Load/save**

Load an existing LDF, or save the one that is currently editing.

* **Undo/redo**

RT

* **Clear all**

Users can clear all the manipulations/changes made on the canvas.

**Alternative texture table**

* **Texture skins & dependencies**

Users are able to change the texture images of walls and other elements, as well as skins of heroes/players. Our game editor, via the very texture table, will provide this.

Extension (alternative)

If possible, users can import their own images other than using system default textures merely.

**Toolbox**

* **Friendly interface**

Our game editor will provide a section that includes all of the editing tools for users for instance, brushes (different size), eraser, etc.

**Preview tab**

* **Preview window**

Users may have a preview of what the map will be like when it actually runs in Bomber man – x.

**AI system**

**Thinking**

* We add a brain for every enemy. If the character is in the attack range of the enemy, the enemy will place bomb and run. If the bonus is in the

**Path finding**

* We use floodfill algorithm to judge the distance between the enemy and the character.

**Physical Engine**

**Particle simulation**

* This part is based on the particle system by Ogre mainly to simulate the real explosion.

The bomb effect can be changed according to the type of the bomb.

**Collision detection**

* We use technology of bounding box. This limits the character’s movements and return a message once detected.

**3D Animation**

**character**

* We provide two kinds of animation for the character.

**Render System**

**Light & Shading**

* We call the functions of the SceneManager class of Ogre.

Different types of point such as Point lights\Spotlight\Directional lights help to make a wide variety of game scenery.

**Shadow**

* We want to implement the depth shadow mapping technique.

**Sound editor**

**Sound effect**

* **BGM**

Choose specific background music of current level.

* **Sound effect**

For instance, Bombs – explosion sound; Heroes/Players – movement/step sound effect, etc.

**Volume controller**

* Users can change the volume of both sound effect and BGM.

**User interface**

**Menu Scene**

* Users can choose the level of the game.

**Options**

* The settings: the sound setting, etc.
* State buttons: Pause/Continue, BackToMenu/BackToGame, Quit, etc.

## Reference

* <http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Tutorials>

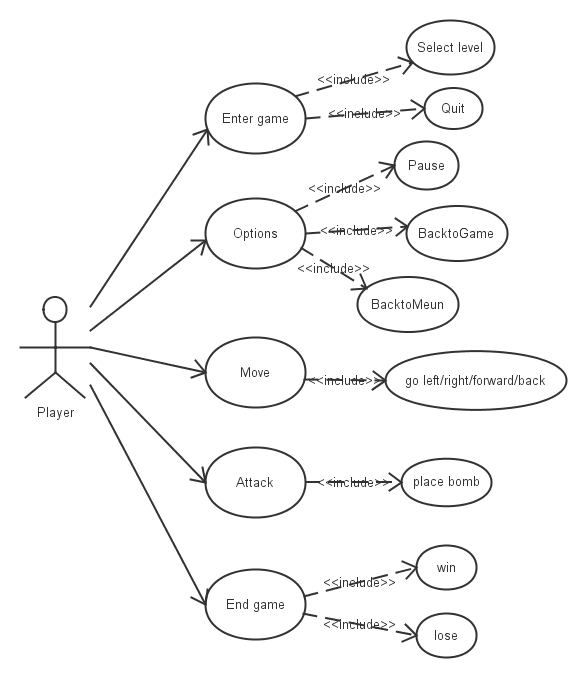
# Implementation details

<This is the meat of the TDD. It includes whatever you need to describe this feature. It can be as long as it needs to be. Feel free to include, suda-code, real code, UML, whatever you need along with paragraphs of text describing how the feature will be implemented. Obviously on smaller features and in smaller TDDs this section might be really small as well.>

**Developing environment**

* Microsoft Visual Studio, OGRE
* Language for game editor: C#
* Language for game : C++

**UML(use case diagram）for functional requirements**



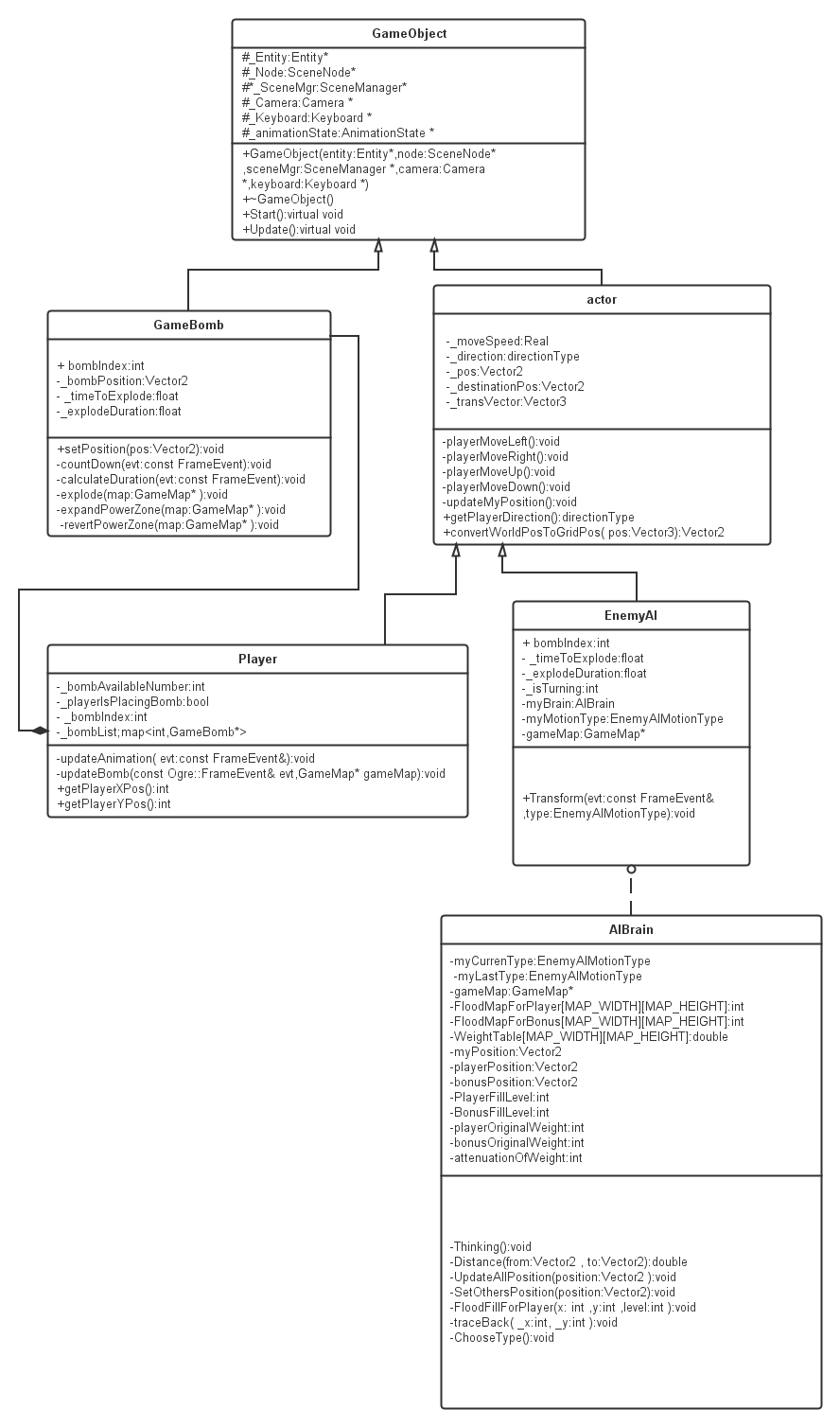
**Character Controller**

|  |
| --- |
| **Character** |
| +Character(3D Entity)  +Actions |
| movement |

|  |
| --- |
| **movement** |
| +Target  +Up  +down  +left  +right |
| Translate  Rotate |

**Object System**

**UML(class diagram）**



**A. character**

Two important elements:

Direction: record the direction where the character goes (the next grid)

Position: record which grid the character is in (Don’t forget to convert world position to the grid position.)

Always update the state of each character.

1. protagonist

The protagonist has a pocket to contain bombs. Don’t forget that the number of bombs is limited.

2. enemy

Each enemy has a brain to tell them how to do next. The judgment is based on the distance between the enemy and the protagonist.

**B. bomb**

Use a time counter to record the time. Once the count equals zero, the bomb will blast.

Get the position of the bomb and find which grid it is in and change the state of the grid as long as it’s in the range of explosion.

The power decides the size of the range.

**AI System**

**AI brain**

According to the map, the brain will calculate the distance between this enemy and the protagonist. If the distance is smaller than a certain number, the enemy will change into the attack mode.

If the distance is larger than a certain number, the pathfinding method will help to find out the next movement for the enemy in a weighted map.

if(distanceToPlayer<minDistanceToPlayer||motionMode==AI\_ATTACK )

{

ThinkingForAttacking();

}

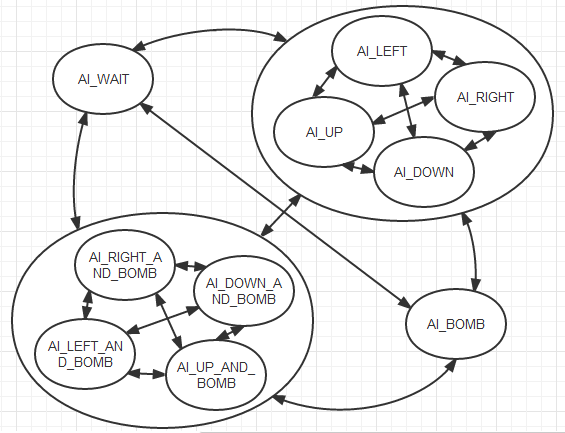
else

{

ThinkingForMoving();

}

* **AI state**



* **PathFinding**

Method: Floodfill

1. The first step is to drop a drop at the starting position and then expend this drop in the whole network until you reach the goal.

Function: FloodFillForPlayer(myPosition.x,myPosition.y,1);

1. Once the goal is reached you find the path by following the track backward from the goal to the start.

Function: traceBack(playerPosition.x,playerPosition.y);

\*Use the stack instead of recursion to effectively solve the problem of lack of memory

Two elements which affect the movement: the character and the bonus.

**EnemyAI**

This is the body for the enemy as a game object.

**Game Editor**

Type: External editor

level data format: .map(the format which we define ourselves and is similar to the txt file)

language: C#

**Physical Engine**

**Particle simulation**

* We use the Ogre::ParticleSystem to implement the particle.

The program will instantiate a particle system when the bomb blasts. The method setParticleAtGrid() in GameMap will be called.

Every new-generated particle system will be named by the time by using the assumeTime(a global time counter) so that to avoid duplication.

When the explosion is over(\_explodeDuration=0), the revertPowerZone which is in the Update method of the GameBomb object, will call the function destroyParticleAtGrid which is responsible for destroy the particle system.

**Collision detection**

* Compare the position of the character and the target to judge collision.

Function: CheckCollision(GameMap\* gameMap)

**3D Animation**

* Ogre::ParticleSystem

Function: updateAnimation(const Ogre::FrameEvent& evt)

**Render System**

**Light & Shading**

* For instance directional light

Ogre::Light\* directionalLight = mSceneMgr->createLight("directionalLight");

directionalLight->setType(Ogre::Light::LT\_DIRECTIONAL);

directionalLight->setDiffuseColour(Ogre::ColourValue(.25, .25, 0));

directionalLight->setSpecularColour(Ogre::ColourValue(.25, .25, 0));

directionalLight->setDirection(Ogre::Vector3( 0, -1, 1 ));

\*Since directional light is supposed to come from a far-off distance, we do not have to set its position, only its direction. We'll set the direction of the light to be in the positive z and negative y direction.

**Shadow**

* The SceneManager of Ogre uses additive stencil shadows.

Ogre::Entity\* entNinja = mSceneMgr->createEntity("Ninja", "ninja.mesh");

entNinja->setCastShadows(true);

mSceneMgr->getRootSceneNode()->createChildSceneNode()->attachObject(entNinja);

**Sound editor**

|  |
| --- |
| **SoundPlay** |
| +Target(WAV)  +Volume |
| Wave Player |

**Naming principles**

* all the name is named by English to avoid garbled data
* avoid using the space
* use “\_”instead of “-” while naming
* use the same naming principle for the same kind of elements

**Memory Implications:**

<How will this effect Memory, it may not>

**Performance Implications:**

<How will this effect Performance, it may not>

|  |
| --- |
| **Hint –** <These things may be needed to think. >  Thread usage (single/multi-threaded)  Supported platforms (PC, 360, PS3, PSP, WII…) |

# Proof

* <Prove the system is working correctly. You can design test case, unit test here. You can also prove it by description, values, or other ways.>

<Example1 – math optimization>

< Unit Test – write unit test for sqrt function… >

<Example2 – Shadows>

< Test Case - We can see smooth shadow on ground that casted by cars and dynamic objects... >

<Example3 – Bullet optimization>

< Verify - After bullet optimized, the function World::stepSimulation() calling time will be in 14ms. >

# Issues

* <Bullet points on any issues for this task.>

# Risks

* **Risk:** The integration of different people

**Mitigation:**

Using a revision control system such as SVN for immediate communication

* **Risk:** Lack of time

**contingency plan:**

To give up some part such as the scene beautification to meet the schedule

# Estimates

<Tasks should be broken into as small pieces as makes sense. Normally no task can be completed in under half a day (considering check in times and building/syncing ect). If a task is more than a day though we should try to break it down into what will be tackled on day vs the next. This being said sometimes a task is 3 days long with no way to break it down more than that. Although this shouldn’t be the norm it does happen and is acceptable. If something is 5 days long make sure that more research isn’t necessary to break it down further. Its often a sign that not enough research has gone into a task if it can’t be broken into smaller pieces than a week, though it can happen.>

|  |  |
| --- | --- |
| **Tasks** | **Estimate in days/person** |
|  |  |
|  |  |
|  |  |
| **Total** |  |

|  |
| --- |
| **Note – Try to make document simple, clean and focus on important parts.** |