Project Proposal for CG 100433 course

Project Title

BomberMan（single version）

Team member

student number	name
1851850	张天昊
1950484	刘家萱
1951727	史睿琪
1953620	高琰婷
1954263	宋子铭

Abstract

In this project, we use CG techniques to implement 'BomberMan', a classic game which is suitable for project's requirement. It is hoped to complish basic game logic with consice UI design and fine CG effect. Using related CG techniques like 3D model loading, multiple light rendering, shadow mapping based on SDL and openGL, we achieve our goals finally.

Motivation

The idea comes from our childhood game —— BomberMan. Every team member is familiar with this classic game and we are excited to complish it in our own way using knowledge from this class. Besides, easy game logic and moderate complexity are in our ability. Numerous open-source models related to this game also help us a lot.

The Goal of the project

1. Achieve players' movement and basic posture.
2. Achieve bomb's placing and explosion
3. Player can compete with enemies, and the game can automatically give a result according to game situation.
4. Achieve the basic light effect, like shadow of player and enemies, objects
5. Achieve shape change of bombers and objects under explosion and light effect of explosion
6. Removable perspective implementation

The Scope of the project

1. loading 3D model
2. scene modeling and map generation

3. multi-source rendering
4. light movement
5. physic engine will not conclude
6. shadows of models

Related CG technique

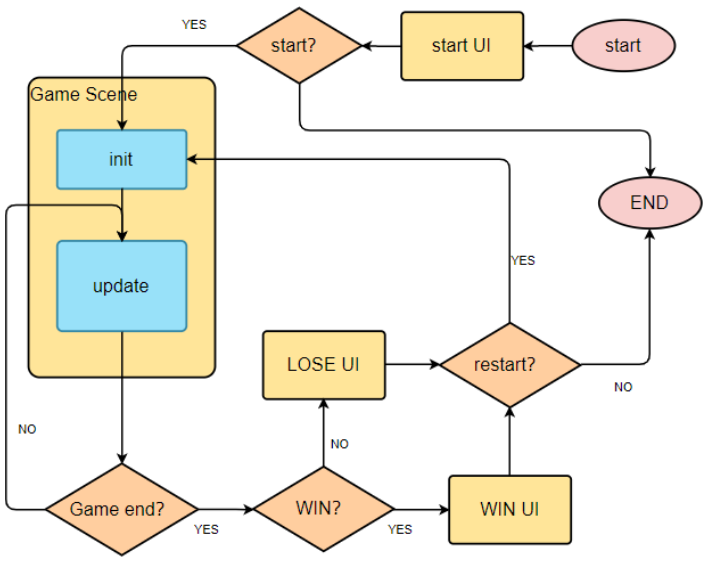
1. modeling
2. viewing
3. transformation
4. rendering
5. shadow mapping

Project contends

1. start/win/lose UI interface
2. map generation, using keyboard to control player.
3. movement of player and enemies. Both only can place a bomber in one square.
4. After specific period, bomber will explode, player or enemies within explosive area will dead, the objects will be destroyed.
5. Pathfinding algorithm is included for enemies' movement. enemies also can put bombers automatically.
6. Decision of the game: Players win only when all the enemies died because of bomber explosion. Conversely, if player dies of explosion, game over.
7. The effect of shadow of objects, multisource light.
8. The effect of bomber explosion.

Implementation

Game logic:



whole process

1. Operation logic:

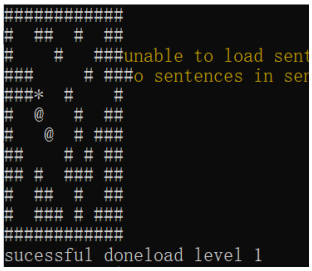
   There are three enemies each round, and you should control “up”,” down”,” left” and “right” keys to let your character move to avoid the bombs these enemies put. The player also can press the “space” key to put a bomb. Every bomb will explode within two seconds after it is placed. When a bomb explodes, any characters around the bomb will be killed, if you can kill all enemies, you win, but if you are killed by any of the bombs, you lose.

2. Implementation logic:

   A total of three classes are used in the completion of the game logic, the player class, the enemy class and the map class. They have different divisions of labor but are interconnected to form a whole game implementation. The game class is responsible for the control of the entire game. It contains the instantiation of the enemy class and the map class, records various real-time information of the player, and provides an interface for interaction, and provides the necessary parameters for real-time rendering of the model and screen. The design also considers the continuity processing required when rendering the enemy's movement, such as providing a bool value of whether an enemy is walking, the direction of walking, etc.

3. Interface interaction logic:

   As mentioned above, the game logic part provides the necessary parameters for real-time rendering models and screens. Among them, the continuous walking of the character is an important problem that needs to be solved. The underlying game logic can realize the automatic pathfinding of the enemy, and update it in real time every second, as showed follow,



   but for the display of the interface, we need to let the enemy slowly moves towards the next position. Therefore, the underlying game logic must meet the function of predicting the position, which means that when a new position is reached, the next position to be reached must also be determined. Then information about whether to walk and the direction of move are given to the drawing part.

Lighting and shadow:

As for lighting, using multiple lights (containing 1 directional light, 10 point lights, 1 spotlight) in a scene the approach is as follows: we have a single color vector that represents the fragment's output color. For each light, the light's contribution to the fragment is added to this output color vector. So each light in the scene will calculate its individual impact and contribute that to the final output color.

```
// define an output color value
  vec3 output = vec3(0.0);
```

```
// add the directional light's contribution to the
output
output += someFunctionToCalculateDirectionalLight();
// do the same for all point lights
for(int i = 0; i < point_lights_number; i++)
    output += someFunctionToCalculatePointLight();
// and add others lights as well (like spotlights)
   output += someFunctionToCalculateSpotLight();
```

Then based on this, add shadow mapping.

The first pass requires to generate a depth map. To generate the depth map simply, we only use one directional light to generate shadow, and moving this light's position to achieve the effect of light movement. Because based on directional light, we use an orthographic projection matrix for the light source where there is no perspective deform.

Next, rendering to depth map with onto a quad.

Finally, with a properly generated depth map, rendering the actual shadows. We do the light-space transformation in the vertex shader, check if a fragment is in shadow is (quite obviously) executed in the fragment shader.

PCF is also added to improve shadow maps.

For combing multiple source and shadow mapping, we choose weighted mean based on visual effect.

SDL control:

The whole game uses SDL management window to process input and output. There are two parts involving input and output in the game. The specific implementation ideas are introduced below.

In the first part, click the UI to jump the control logic.

Since the jump logic has been introduced in the previous section, only the trigger of mouse click event is described here. Because the interface provided by SDL can only detect the mouse click position and return the (x, y) coordinates of the position in the window, it does not provide controls such as "buttons" that can be used to trigger events. Therefore, for convenience, we adopt a more ingenious method.

Instead of realizing the real button control, we drew a map on the whole screen with OpenGL texture technology. The proportion of this map is the same as the screen size to ensure that it looks like a complete interface with rich controls (the effect can be seen in the results section).

Next, we use the following method to simply and directly specify a part of the interface as a "button".

As shown in the following code segment, we surround a rectangular area with up, down, left and right. When the mouse click position falls within this rectangular area, the corresponding event can be triggered, thus realizing the function of the control.

```
while (SDL_PollEvent(&e))
{
    if (e.type == SDL_MOUSEBUTTONDOWN && SDL_BUTTON_LEFT == e.button.button) {
        int px = e.button.x;
        int py = e.button.y;
        //printf("x, y %d %d ..............\n", px, py);

        if (py >= up_border && py <= down_border) {
            if (px >= start_left && px <= start_right)
                _states.newGame = true;
            else if (px >= end_left && px <= end_right)
                _states.exit = true;
        }

    }
    else if (e.type == SDL_QUIT) {
        _states.exit = true;
    }
}
```

Another part of SDL processing input is keyboard control operation. This part is relatively simple and can be judged directly by the key events provided by SDL, as shown in the following code:

```
void Player::OperateBySDL() {
    SDL_Event ev;

    while (SDL_PollEvent(&ev))
    {
        if (SDL_KEYDOWN == ev.type) // SDL_KEYUP
        {
            POS moveTo = this->pos;
            if (SDLK_DOWN == ev.key.keysym.sym)
            {
                moveTo.x++;
                dir = DOWN_PMOVE;

            }
            else if (SDLK_UP == ev.key.keysym.sym)
            {
                moveTo.x--;
                dir = UP_PMOVE;
            }
            else if (SDLK_LEFT == ev.key.keysym.sym)
```

Model rendering:

In the model rendering part, because we have the underlying state array, generally speaking, model rendering is to display the model on the interface according to the

underlying array.

For example, the status array provides the information of the currently existing objects (walls, bombs, players, enemies, boxes) in the block, and provides the moving direction of the movable object, so as to move the object slowly.

The more complex implementation here is the mobile enemy. Because the moving direction of the enemy is given in the state array, if we can render continuously, we can get a complete moving process.

At the beginning of this continuous rendering, record a time to prepare for rendering, and then draw. It should be noted that each time you draw, first calculate the orientation of the model, and use the rotation matrix to change the model state. After the change is completed, the offset of the model based on the current position is calculated according to the recorded time.

The specific formula is as follows:

① Define a current location (curx, cury) for each moving model

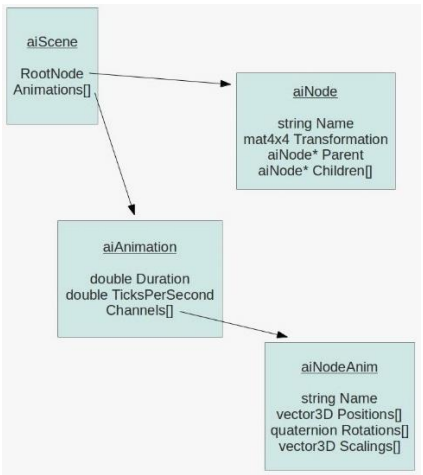
② When the position of the transmitted dynamic model parameters (x, y) is the same as the recorded curx, cury, use (current time - startms) * speed + POS (add in direction)

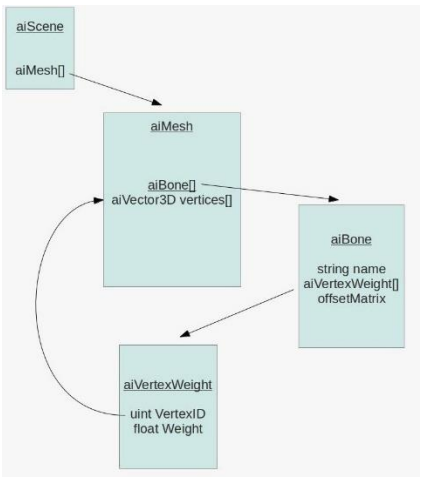
③ When the position of the parameters (x, y) transmitted from the East model is different from....., reset the three parameters and draw the model on the new x, y

Skeletal Animation

To bring our game to live, we managed to implement skeletal animation during model rendering. Obtained from the internet, the models have stored within the files themselves information about its skin(meshes) and bones. The bone structure used for skeletal animation is hierarchical. When a parent bone moves it also moves all of its children, but when a child bone moves it does not move its parent. In every node in the bone tree there are a scaling vector, a rotation quaternion and a translation vector. In time of the application, we interpolate the scaling/rotation/translation to get the correct transformation for the point, and do the same process for each node from the current bone to the root and multiply this chain of transformations together to get the final result. We do that for each bone and then update the shader.

The data structures about model animation in assimp library are as follows.





In the rendering loop, we pass a time parameter to the function that updates current model transformations with interpolation performed, since the animation sequence provided by the model only contains information corresponding to key frames. The results are stored in a vector of transformation matrices, and are then passed to model-rendering shaders.

Interpolation functions:

```
void calcInterpolatedPosition(aiVector3D& out, float animationTime, const aiNodeAnim* nodeAnim);
void calcInterpolatedRotation(aiQuaternion& out, float animationTime, const aiNodeAnim* nodeAnim);
void calcInterpolatedScaling(aiVector3D& out, float animationTime, const aiNodeAnim* nodeAnim);
```

Get the transformation vector:

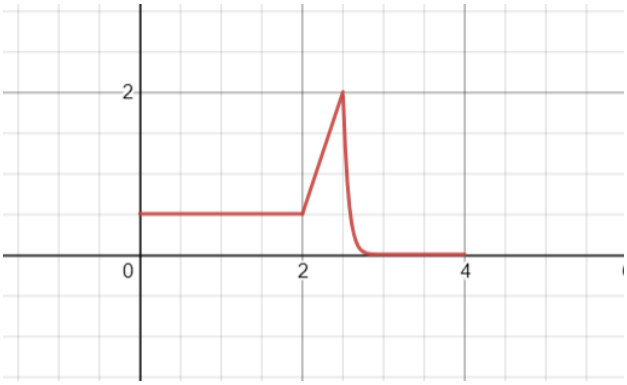
```
void Model::boneTransform(float timeInSeconds, std::vector<glm::mat4>& transforms)
{
    glm::mat4 identity = glm::mat4(1.0f);

    //animation duration
    unsigned int numPosKeys = scene->mAnimations[currentAnimation]->mChannels[0]->mNumPositionKeys;
    animDuration = scene->mAnimations[currentAnimation]->mChannels[0]->mPositionKeys[numPosKeys - 1].mTime;

    float ticksPerSecond = (float)(scene->mAnimations[currentAnimation]->mTicksPerSecond != 0 ? scene->mAnimations[currentAnimation]->mTicksPerSecond : 25
    float timeInTicks = timeInSeconds * ticksPerSecond;
    float animationTime = fmod(timeInTicks, animDuration);
    readNodeHeirarchy(animationTime, scene->mRootNode, identity);
    transforms.resize(bonesCount);

    for (unsigned int i = 0; i < bonesCount; i++)
        transforms[i] = boneMatrices[i].FinalTransformation;
}
```

Dealing with bone transformation in vertex shader:

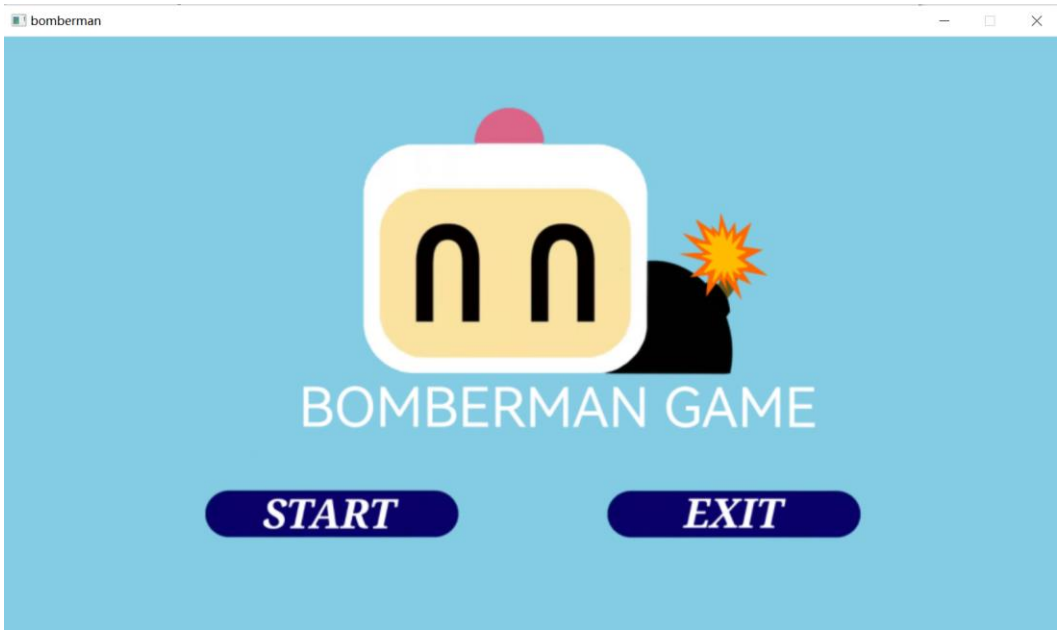
```
if (animated)
{
    mat4 BoneTransform  = gBones[aBoneIDs[0]] * aWeights[0];
        BoneTransform += gBones[aBoneIDs[1]] * aWeights[1];
        BoneTransform += gBones[aBoneIDs[2]] * aWeights[2];
        BoneTransform += gBones[aBoneIDs[3]] * aWeights[3];

    vec4 tPos = model * BoneTransform * vec4(aPos, 1.0);
    gl_Position = projection * view * tPos;
    FragPos = vec3(tPos);
}
```

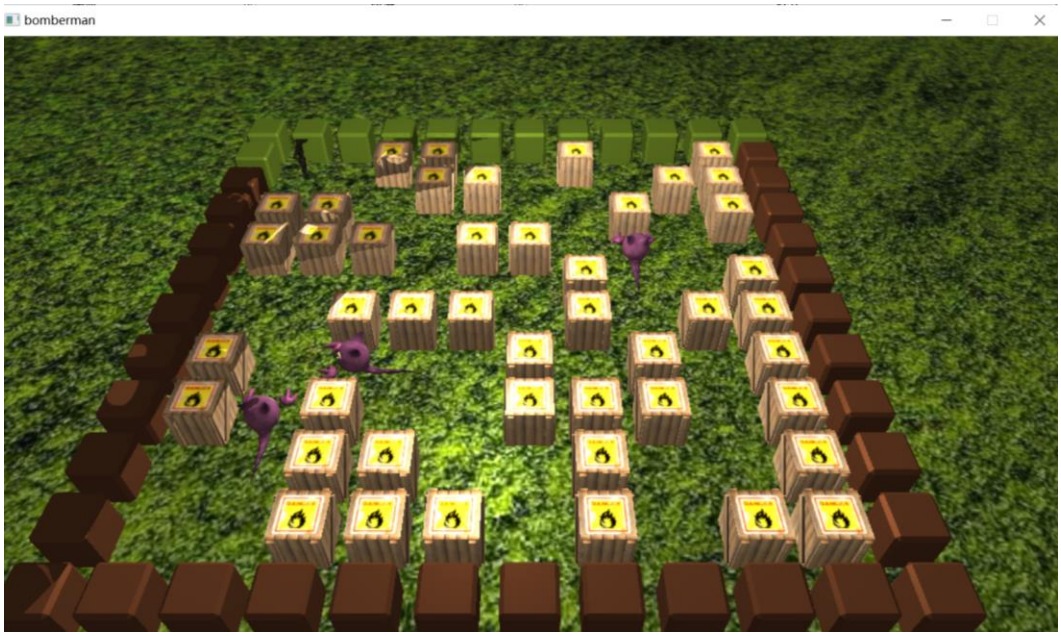
Bomb explosion effect:

When each bomb is generated, we will assign a light source to the bomb. Since the light source is defined in the shader, the number is fixed and has a unique number, we manage these light sources in a stack. Initially, the light intensity of the light source is 0, so it looks like it doesn't exist. When it is assigned to the bomb, it will change the light intensity according to a functional law according to the time record. This function law looks like the light effect produced by explosion, and its curve is roughly as follows:



Results





Roles in group

work	name
Game logic (Find Path algorithm of enemies, decision of bomber placement, map generation, judgement of game)	高琰婷
Model import & texture	刘家萱
dynamic model animation	刘家萱，史睿琪
Skybox implementation	史睿琪
UI design	张天昊，史睿琪
The effect of explosion Project connection	张天昊
Game logic (changing interface logic, Gui design) The effect of shadow multiple light rendering & light movement	宋子铭

Reference

- LearnOpenGL CN (learnopengl-cn.github.io)
- https://github.com/qwikdraw/bomberman
- tnicolas42/bomberman-assets at 85f00248d78a58e6318ebca8601b4e9f3295ff28 (github.com)
- Q 版泡泡堂小游戏,在线玩,4399 小游戏