Pokemon Fight! Pikachu VS Snorlax

Name: Zhenyan Zhu
Student ID: 20817494
User ID: z277zhu

December 2, 2022

1 General

The purpose of this project was to implement a extendable 3D Pokemon battle game. I feel that I have successfully met all of my goals and even add other extra features to make this an complete game.
This game requires you to control Pikachu and defeat the Boss - Snorlax. You should circumvent from Snorlax's attack, including Body Slam and Meteorite fall and attack Snorlax by discharging.

2 Project Structure

/docs : project report and proposal
/lib : static and shared third party libraries
/shared: source code of third party libraries
/src: source code of project
    /src/Assets : shaders, textures, audios, lua modelling, .obj files
    /src/include: include files (*.hpp) of project
    /src/src: implementation files (*.cpp) of project
    /src/modelling: modified A3 source code for collecting animation assets

3 Code Mapping

1. Animation system (src/src/Animation.cpp, src/include/Animation.hpp)
2. Particle systems that can be used by Lua scripts (src/src/particle.cpp, src/include/particle.hpp, src/src/scene_lua.cpp)
3. Texture importing and caching system that can be used Lua scripts (src/src/texture.cpp, src/include/texture.hpp, src/src/scene_lua.cpp)
4. Shadow (Assets/shadow_depth.fs, Assets/shadow_depth.vs, Assets/Pong.fs, src/src/shadow.cpp, src/include/shadow.hp src/src/GameWindow.cpp)
5. Grass (Assets/grass.fs, Assets/grass.geo, Assets/grass.vs src/src/GameWindow.cpp)
6. Sound (src/src/GameWindow.cpp, src/src/sound.cpp, src/include/sound.hpp)
7. Pokemon behaviors (src/src/GameObject.cpp, src/include/GameObject.hpp)
8. Scene modelling, Pokemon modelling (Assets/Scene.lua, Assets/pikachu.lua, Assets/snorlax.lua)

4 Manual

4.1 Build

On Linux, run 'bash build_linux.sh' at the root of project; on Mac, run 'bash build_mac.sh'

4.2 Run

cd src && ./game

4.3 Game Control

1. **Movement:** The movement of Pikachu is controlled by keyboards. Similar to other games, 'W' controls Pikachu to move up; 'A' controls Pikachu to move left; 'S' controls Pikachu to move down; 'D' controls Pikachu to move right.

2. **Attack:** Press 'j' to use discharge from Pikachu's left hand; press 'k' to use discharge from Pikachu's right hand.

3. **Others:** Press 'q' to quit the game; press 't' to enable/disable toon shading.

4.4 Output

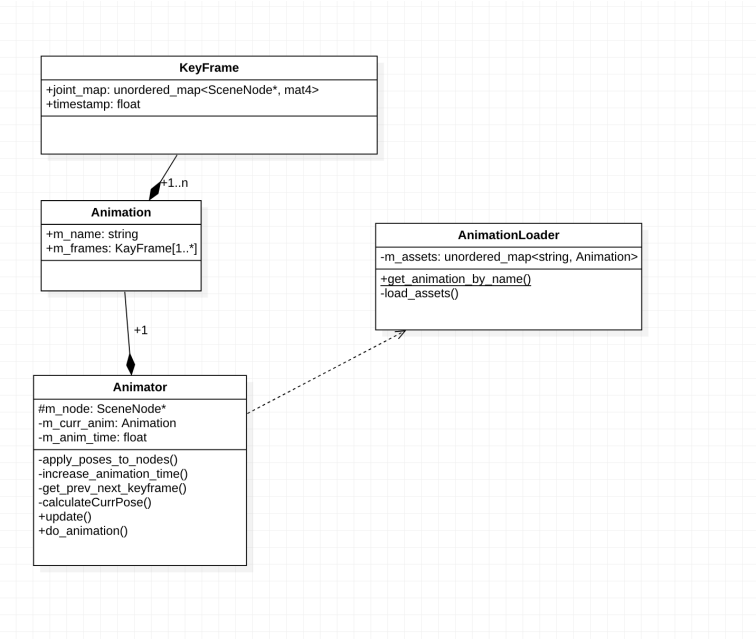
There will be no command-line output when the debug mode is off.

5 Implementation

I listed several interesting technical points that I found most challenging and spent a large amount of time designed and implemented. For other parts, please refer to my project proposal for technical contents.

5.1 Animation System

This section describes the design and implementation of my animation system. (src/src/Animation.cpp;src/include/Animation
The following UML diagram describes a high-level view of the components of the system (some getter and setter are omitted).



I would like to mainly introduce the AnimationLoader and the Animator class.

1. **AnimationLoader**: It is a Singleton that holds the assets of all animations.
   in the beginning of the game, the AnimationLoader will load animations of Pikachu and Snorlax from Assets/*.ani files. *.ani file has the following protocols:
   $animation1Name$ $num of keyframes$
   $num of nodes in frame 1$ $timestamp at frame 1$
   $name of node 1$ $quat.w$ $quat.x$ $quat.y$ $quat.z$ $trans.x$ $trans.y$ $trans.z$
   ...
   $name of node n
   ...
   $num of nodes in frame m$ $timestamp at frame k$
   $animation2Name$ $num of keyframes$
   ...
   All the animations will be loaded into the m_assets field of the AnimationLoader, which is an unorder_map such that we can lookup an animation by name with runtime O(1)

2. **Animatior**: It is the class that perform the updates of an animation.

   (a) To perform an animation, *do_animation(Animation* ani_ptr)* will be invoked. *ani_ptr* is a pointer points to an animation that holds by the AnimationLoader. When set the animation to *m_curr_anim*, despite clean the current animation and copy all the fields from the ani_ptr, it will also fetch the

frame 0, which is the current pose of the nodes defined in frame1 into the current animation for the interpolation calculation in *update()*. It will also set the current time of the animation.

(b) the *update()* method will be called in each frame that calculates the current pose and apply to the all related SceneNodes. It has the following procedures:

i. It will update the current timestamp after do_animation() is called time by utilizing the routines in timestamp.hpp.

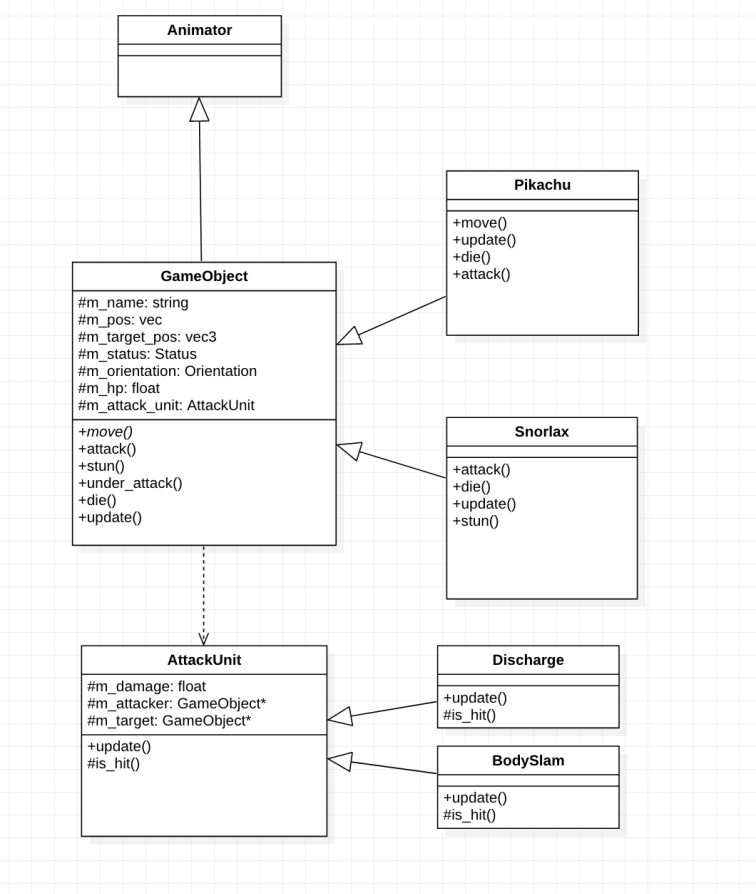
ii. It will fetch the current and next keyframes by calling *get_prev_next_keyframe*, which takes O(1) because the current frame index is maintained.

iii. It will calculate the current poses, which are the transform matrices of each SceneNode that the current keyframe tracks.
We will first calculate the current progression of the current frame, which is calculated by the formula $progression = \frac{(currentTime - prevKey.timestamp)}{(nextKey.timestamp - prevKey.timestamp)}$
Then, we calculate the interpolated transformation matrix for each SceneNode at the current progress by calling *glm::interpolate(prevKey.transforms[node], nextKey.transforms[node], progression)*
Finally, we apply the current poses to these SceneNodes by multiplying the difference between the calculated transformation and the current transformation (*calculatedTrans * inverse(currTrans)*) to their current transformations.

All the fields and methods of Animator are private, and other components can only invoke the update() or do_animation() methods, so it's quite encapsulated.

5.2 Game Objects

This section will introduce the design and implementation of the core component of the game - the game objects including Pikachu and Snorlax. The following UML gives a high-level description of the game objects. (src/GameObject.cpp, include/GameObject.hpp, src/AttackUnit.cpp, include/AttackUnit.hpp)



The abstract class *GameObject* is inherit from the Animator class as we talked above because the GameObject is an Animator that we want to interact with. It is highly abstracted as there are only 5 main methods exposed to other components:

1. *move(x,z)*: this method will not directly move the game object with current position + (x, 0, z), but it will only set the *m_target_pos* field to be the target position and delegate the work in the *update* method. Pikachu's move will do the corresponding animation; Snorlax cannot move because his BodySlam will help him to move

2. *attack(name, target)*: this method will instantiate an AttackUnit object and attach it to the *m_attack_unit* field and delegate the work of the real attacking in the *update()* method of this attack unit.

3. *stun()*: this method will be overwritten by Snorlax, after he did BodySlam, stun() will be invoked and he will do the stun animation.
   Pikachu does not have stun()

4. *under_attack(damage)*: this method will set the hp of the game object to be $max(0, m_hp - damage)$, when it's lower than 0, it will invoke the die() method and notify the game window to stop the game.

5. *die()*: this method will set the status of game object to be Status::Dead such and perform the corresponding side effects. For Pikachu, it will be smashed into a pancake. This can be done by scale Pikachu's SceneNode with a scaling factor of (1,0.01, 1); For Snorlax, a die animation will be played.

6. *update()*: this method overwrites Animator::update(). It will invoke Animator::update() when *m_status* is not idle(which means the object is performing some tasks with animations).

Meanwhile, it will also invoke the update() method of m_attack_u to update the attack unit. Additionally, it will interpolate the position of the game object when its target position is set.

7. *AttackUnit::update()*: this method defines the main logic of how a GameObject attack another. Concrete AttackUnit has their own implementation of update.

   (a) *BodySlam::update()*: When BodySlam is instantiated, it will set the attacker's target position's y component to SKY(100.0f) such that the *GameObject::update()* will eventually move the object to the sky. the update() method has two phases and it will query the position of the attacker in each frame. In phase1, if the attack unit found that the game object has touched the sky(pos.y == 100), then it will perform phase2() by setting the x and z components of the attacker's position to the target object's position, and set the attacker's target position's y component to the GROUND(y=-8). When it found that the game object touched the ground, it will emit the particle effect of flying dirt, shake the camera and invoke the *under_attack()* method of the target object if hit.

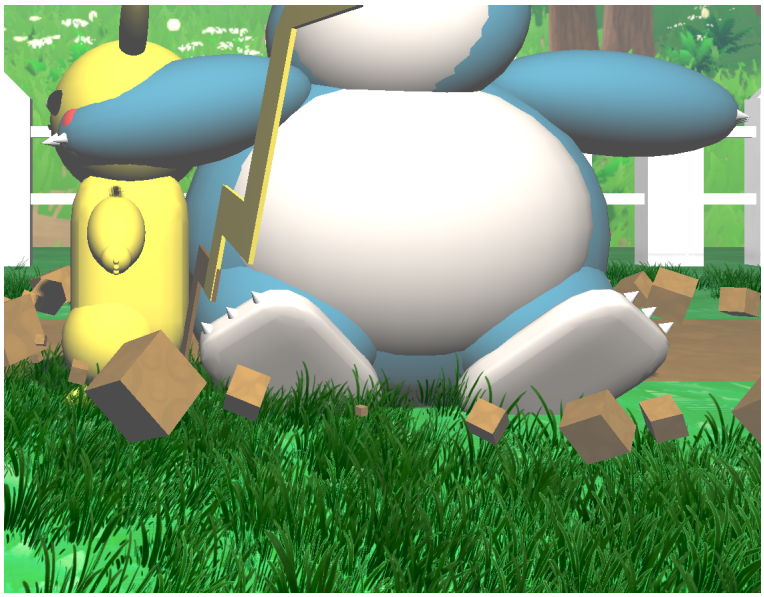
   (b) *Discharge::update()*: The discharge will hold a remaining time for instantiating another discharge effect. In each update(), the remaining time will be updated and when it's less than 0, it will invoke *lightning_effect()* to discharge lightning. This forms the effect of the a realistic discharge because. In each frame, the attack unit will also determine whether the ray(lightning) hits the target object or not by determine the intersection of a ray to a sphere as we implemented in A4 and generate damage to the target object.

5.3 Particle Systems & different particle effects

This section describes the structure of particle systems and how I implemented different particle effects including flying dirt, lightning and meteorite fall. (/src/particle.cpp, /include/particle.hpp)

1. The ParticleSystem holds a pool of Particles and a index points to the next available particle encapsulated. The other component should invoke the *Emit* method to emit a new particle.
   Each Particle has their own position, velocity, rotation, rotation velocity, size, gravity and lifetime with some random noise factors. In ParticleSystem::update(), all the active particles will update their position by their velocity and update their velocity by the gravity to achieve the realism.

2. *dirt_flying_effect(radius, position, base_num)*: After invoking this routine, you would see random numbers of dirts,ranging from base_num to 1.5 * base_num,spilled out from the ground at the given position forming a circle with the given radius, then falls onto the ground after a while. Effect is shown below.



To implement this effect, we first calculate the angle k difference between two dirt square $k = \frac{2\pi}{numberofsquares}$. Then, for $square_i$, the angle is $\phi = i * k$, the x position is position.x + radius * $cos(\phi)$, the z position is position.z + radius * $sin(\phi)$, the velocity will be normalize($square_i$ - position) + vec3(0,10,0) to make the squares have an effect of spilling out from the given position.

3. *lightning_effect(position, dir)*: After invoking this routine, you should see a lightning beam emitted from a given position with some randomly generated electrons nearby.

This beam is implemented mainly by 3 parts. The first part is the trunk of this beam, which consists of a random number of cubes with a random rotation from -30 to 30 degrees on 'x' axis; the second part is the branches of the beam, which consists a random number of cubes with a random rotation from -90 to 90 degrees on 'x' axis; the third part is the electrons, which consists a random number of small squares with some random offset to the trunk. The beam will have a short lifetime (0.1s), such that the Discharge::update() will invoke this method at a high frequency to achieve realistic lightning effect.

4. *meteorite_fall(base_num)*: this is the routine that generates base_num to 1.5 * base_num of meteorites that will falling from the sky to the ground.

This is implemented by randomly generate meteorite particles at x position from -50 to 50, y position from 30 to 150, z position from 0 to 100. Each meteorite will also be attached with a destroy function. When the meteorite is destroyed (touched on the ground), the destroy function will be invoked and the following side effects will be performed: a flying dirt effect will be emitted at given position, if the meteorite hits Pikachu, the under_attack method of Pikachu will be invoked to lower the hp. A camera shake will be performed.

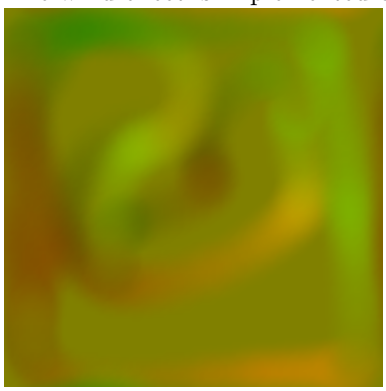
5.4 Realistic Grass Rendering

This section will discuss the implementation of realistic grass utilizing geometry shader. The high level idea of the implementation contains three parts

1. We randomly generate points of the grass in the scene and pass to the vertex shader (grass.vs) using GL_POINTS. Note that the vertex shader will not multiply the View and Perspective matrices to the points as usual, but it will directly pass the point to the geometry shader because the modelling of grass will be implemented at geometry shader's side(grass.geo).

2. The geometry shader will take the point passed by the vertex shader and construct 3 crossed quad given that position. The cross effect is implemented by rotate a quad by 45 degrees of y axis. The creation of quad will also generate the uv coordinates of the each vertex of the quad that will be passed to the fragment shader for texture mapping.

3. The wind effect is implemented by texture mapping of the flowmap.



The red value of the image represents the direction in x, the green value of the image corresponds to the direction y, and the uv coordinates maps from the quad coordinates to the flowmap is calculated by the formula $uv = mod(\frac{base\_position.xz}{100.0} + windStrength * Time, 1.0)$, where the windStrength is 0.1f, Time is the current time queried by *glfwGetTime()*. This formula guarantees that grass at different position and at different timestamp will have different wind effects, which achieves the realism.
Moreover, a 2D noise is added to the input of fragment shader to interpolate the color of grass from dark to light to achieve realism.

4. The fragment shader will apply the texture

to each vertex emitted by the geometry shader. Note that there are some transparent part of this texture, so the fragment shader will discard the color if the 'a' component of the color after texture is less than 0.25. It will also interpolate the color between color and 0.5 * color by the noise factor passed by the geometry shader through the *mix()* routine.

This is the final effect of the grass:

5.5 Additional features beyond the objective list

To make this game complete, I also add some extra features:

1. There is a start menu to let the user check the control manual, select the sound track and start the game. This is implemented by imgui

2. There are hp bars for Pikachu and Snorlax indicating current hp of Pikachu and Snorlax. This is implemented by 3D rendering without Phong shading. When Pokemon are under attack, their related hp bar will be scaled and translated to achieve the effect of hp bar.

3. Camera shakes are implemented to add the excitement of the game. This is completed by adding a noise to the original view and updates the view every frame. After the shake is over, the original view will be restored.

5.6 Lua Extension

1. **Texture:** to import a texture in Lua, you should write *gr.texture("file.png")*, where file.png should be located in "src/Assets/textures"; to apply a texture to a geometry node, you should write *node.set_texture(texture)*
2. **Particle:** to add a particle system that emits particles using a geometry node, you should write *node.set_particle()*
3. **Deep Copy:** to tile the scene with multiple geometry nodes, I developed a method named node.add_child_deepcpy(anoth such that it will perform a deep copy on the other node instead of directly points to it. This greatly boost my development of the scene because I can use a loop to add children into the scene.

6 Others

6.1 Design Patterns & Code

1. Singleton and Visitor design patterns are utilized in this project to improve the maintainability and extendability.

2. For objects that may traverse the SceneNode to perform some tasks, such as the GameWindow that will traverse the SceneNode to render the meshes, they only need to overwrite the visit(SceneNode*) method for a concrete SceneNode, then the recursive part happens automatically.

3. For objects that only appear once in this project and required to be accessed by multiple components, such as the Player class, Singleton pattern is utilized to lazily initialize the instance and offer interface for other components to access.

4. All the code are well commented and assertions are utilized completely to make sure the program executes correctly.

6.2 Debug Utilities

in 'src/include/debug.hpp', I implemented some utilities for better debugging, if NDEBUG is defined by make config=release, DLOG and PRINT will have no effects; if NDEBUG is not defined (make config=debug), DLOG will perform the same to printf, while PRINT macro is for us to incorporate some debug statement, such as cout statement to print some objects like mat4 or vec3.

```
#ifdef NDEBUG

// disable functions
#define DLOG(...) ((void)0)

#define INIT_LOG
#define PRINT(stmt)

#else

#define DLOG(fmt, ...)                                                  \
    do {                                                                \
        fprintf(stderr, "DEBUG  [%s:%d] ",__FILE__,         \
                __LINE__);                                              \
        fprintf(stderr, fmt, ##__VA_ARGS__);                            \
        fprintf(stderr, "\n");                                          \
    } while (0)

#define PRINT(stmt) stmt
#endif // NDEBUG
```

6.3 Animation Assets Collection

All animations are collected by the modified A3 program in /src/modelling, when we deselect a JointNode, its quaternion and translation will be printed out.

7 References

7.1 Assets References

All of the textures are free and retrieved from `https://www.textures.com/`
All Pokemon background music are free and retrieved from the free video game music website `https://downloads.khinsider.com/game-soundtracks`
Alll sound effects, including lightning and crushing, are purchased from the website `https://www.tukuppt.com/yinxiaomuban/dianji5781.html`

7.2 Technical References

Learn OpenGL - Graphics Programming, de Vries, Joey, 2020.
I use the web version at `https://learnopengl.com/` here are some specific articles:

1. `https://learnopengl.com/Guest-Articles/2020/Skeletal-Animation` for key-frame animation
2. `https://learnopengl.com/Advanced-OpenGL/Cubemaps` for cube maps and texture mapping
3. `https://learnopengl.com/In-Practice/2D-Game/Particles` for particle systems
4. `https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping` for shadow mapping
5. `https://learnopengl.com/In-Practice/2D-Game/Audio` for 3D audio

GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Hubert Nguyen, 2007.
I use the web version at `https://developer.nvidia.com/gpugems/gpugems/contributors`, and I learnt netural grass effect under the article `https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-7-rendering-countless-blades-waving-grass`

8 Acknowledgement

I would like to acknowledge that the 3D sound library utilizes the irrklang library, download at the official website `https://www.ambiera.com/irrklang/downloads.html`
I would like to acknowledge that the 2D noise function used in src/Assets/grass.geo is based on `https://www.shadertoy.com/view/4dS3Wd` by morgan3d

Objectives:

Full UserID: z277zhu **Student ID:20817494**

___ 1: There is an environment mapping utilizing cubeMap to render a Pokemon skybox and a texture system to load and cache textures.

___ 2: There is a 3D sound system that can both generate the background music and provide sound effect in the battle

___ 3: There is a series of different animations utilizing transformation when the Pikachu and Snorlax are running, standing and attacking;

___ 4: There are particles simulating lightning and flying dirt when the two Pokemon attack

___ 5: Toon shader is added, and there will be a button to switching from Toon shader or Phong shader.

___ 6: Shadow mapping is added, and there will be a button to turn on/off the shadow mapping.

___ 7: An artistic Pokemon scene will be rendered

___ 8: realistic grass with wind effect is modeled by texture mapping and geometry shader

___ 9: Snorlax AI is added

___ 10: The Pikachu model in A3 will be upgraded and a Snorlax model will be created to support different motions.

I would like to mention that for Objective 10, I planned to add a hip to Pikachu previously such that when the Pikachu is moving, he would transform from standing on two legs to standing on four legs, and he would move forward with his four legs; however, when I implemented this, I found it extremely unnatural, so I removed the hip and add more details to the Pikachu's back.