Joust

Game Engine Architecture, DIT455

Christopher Black

March 2019

Figure 1: A picture of the game Joust. The arena has a fixed layout. The player and enemy knights moving by flying or walking on the platforms. The pterodactyl enemy targeting the player knight.

1 Introduction

This Report describes the implementation of the 2D game Joust. Joust was originally released as an arcade game in 1982, developed by Williams Electronics [2]. The game features up to two players, controlling a knight riding a bird as mount each. The objective of the game is to score as many points as possible by jousting non-player knights and defeating pterodactyls, while also avoiding stage hazards, in the form of lava and troll hands grabbing for player as well as non-player knights.
The version of this game, of which the implementation is discussed in this report is however not an exact replication of the original. Features included in the original, which are missing from the version discussed here, are the troll grabbing for the knights when they are getting to close, multiple types of enemy knights with different behaviours, platforms disappearing over time and support for a second player. Amongst minor changes, e.g. the enemies' behaviours deviating from the ones in the original, one larger change made to the gameplay is, that instead of re-spawning the player at one of multiple spawning points after losing a life, the player is getting two seconds of invincibility to reposition himself.
In the remainder of this document, the mentioning of Joust refers to this modified implementation of the game, unless it is explicitly mentioned that the original game is being referred to.

2 Specification

The game Joust is made up of five elements: the player knights, the enemy knights, eggs, the pterodactyl and the arena, see figure 1. The goal of the game is to score as many points as possible by defeating enemies before losing all lives.

The player knights. The player knight can fly with quick upward impulses, simulating the flapping of bird wings, and horizontally, by walking on platforms as well as while being airborne. The player knight can attack enemies by colliding with them. The player knight has three lives, which can be lost to enemies or by colliding with hazards.

The enemy knights. The enemy knights have similar movement options as the player knight, this being moving vertically by flying upwards or falling downwards and moving horizontally. They do not target the player knight, but traverse the arena randomly. Enemy knights are defeated, when they are colliding with the player knight, while the player is in a higher position, If the enemy knight is in the higher position the player knight will lose a life.

The eggs. The eggs are items dropped by enemy knights, when they are defeated. They will add to the score, when they are collected by the player knight, or be the destroyed, when they get in contact with the lava. If neither happens within ten seconds a new enemy knight is spawned from the egg.

The pterodactyl. The pterodactyl is an enemy, which is spawned less frequent than the enemy knights. The pterodactyl moves at a constant speed and targets the player knight. To defeat the pterodactyl the player knight has to collide with the it at a certain height in a specific frame of the animation, thus hitting the pterodactyl in it's throat. Other collision with the pterodactyl make the player knight lose a life. Defeating the pterodactyl adds to the score of the player.

The arena The arena stretches across the whole screen and consists of eight platforms and a lava pit at the bottom of the screen. Any knight, pterodactyl or egg exciting the screen horizontally reappears on the other side of the screen, thous four of the platforms are connected in pairs. When hitting the lava, the player knight loses a life and enemy knights are defeated and drop an egg.

The end of the game. If the player knight loses all three of his lives by colliding with enemy knights, the pterodactyl or the lava, the player loses and the game ends. If the player manages to defeat all enemies and collect all eggs on the screen, a new wave of enemies is created.

3 Game Flow

The game starts with the player knight having three lives, a score of zero and being positioned at the lowest platform and four enemy knights and the pterodactyl, each positioned randomly at one of four possible spawning points, which are left and right outside the boarders of the screen. Once all of these enemies are defeated and all of the eggs dropped from the enemy knights have been collected a new wave of enemies is created, with the number of enemy knights increased by one. The maximum amount of enemy knights is fifteen, all further enemy waves will contain fifteen enemy knights. The pterodactyl is only spawned every third wave. The player can increase his score by collecting the eggs dropped from enemy knights, which will reward the player with 250 points, or by defeating pterodactyls, which will reward the player with 1000 points. The game is over, when the player has no lives left.

4 Implementation

In Joust multiple objects interact with each other. These objects are just collections of components with a shared minimal state, containing the position, the id and an optional tag of the object. All functionality of the objects is implemented in components. The object also allows communication between components, by forwarding messages from one component to all of the other components. There are four types of generic components used by multiple types of objects: Render, Sprite Animation, Collide and Rigid Body. The render component is responsible for drawing the 2D sprite representing the game object to the screen. The

sprite animation component changes the displayed sprite at a variable rate. The collide component detects collisions between the object and a set pool of other objects, while the rigid body component is responsible for applying simulated physics on the position of the object, which includes applying gravity and other forces, as well as resolving collisions. All object are updated by the engine. While the engine takes care of generating the objects, the object generation is requested from the main procedure at the entry point of the application, where the object composition takes place. As all the code which is updated in the game loop is either generic engine code or code attached to objects in the form of components, one object is created to keep track of the game state. This object is called game manager. This game manager is responsible for keeping track of the active amount of enemies, spawning the enemy waves, verifying the ending condition and visualising the state of the game on the UI in the form of the player's score and remaining lives. Theoretically this implementation of Joust can be divided into two parts: The generic engine part, which is meant to provide basic functionalities ease game development and be reusable in other projects, and the game specific part, which is meant to utilize the functionalities provided by the engine and take care of the game logic specific to Joust.

4.1 Game Loop

The game loop [7, Chapter 9] is implemented as part of the generic engine, allowing it to be reused in other projects without having to write a new game loop or adjusting the existing one. For the game loop to run it is called once from the main procedure. In one iteration of the game loop, first the time, which elapsed since the start of the previous iteration of the loop, is calculated and stored as delta time. Next the engine is polling for player input. Here the game loop is exited if either the escape key or the close button of the window is pressed. After that all enabled game objects and their components are updated [7, Chapter 10] and then the scene is drawn to the screen. At the end of an iteration of the game loop the difference between the requested milliseconds per frame (in this case 16, resulting in about 60 frames per second) and the milliseconds, which elapsed between the start of this iteration and this instant in time, is calculated for the engine to wait this amount of milliseconds before starting the next iteration to ensure a constant frame rate as long as the requested milliseconds are not exceeded by the computation of the loop iteration.

4.2 Object Pool

An Object pool [7, Chapter 19] is a templated class, which is used to pre-allocate memory for a set of objects of any type. Which type of objects a object pool contains, has to be specified in angular brackets.

```
1 ObjectPool<GameObject> enemies;
2 ObjectPool<GameObject> eggs;
```

These objects are create when the game is loaded and then stored in the object pool. When a a object is not currently used it is disabled instead of destroyed. This way, when an object is needed during the game, the object pool is requested to provide a currently available (in other words one of the disabled) objects. This allows reusing objects, instead of having to create new ones in the game loop, which means having to allocate memory and is computationally costly. When the game is destroyed the objects in the pool are deallocated in the clean up process.
In Joust there are four object pools implemented, all of them containing game objects, but serving different purposes. Two of them containing the enemy knights and the eggs respectively, while the other contain all objects that collide with arena colliders and all objects that collide with the player knight.
The objects contained in an object pool are internally stored in a std::vector [5], which is an array of dynamic size, but allows access to any of its elements in constant time O(1), similar to a regular array of constant size.
Selecting the first available object in an object pool is done in the FirstAvailable() method, which loops through the std::vector and returns the first object, which is disabled.
It is also possible to retrieve a random active (currently enabled) object from the object pool, using the SelectRandom() function. First a random offset between 0 and the current number of elements in the object pool is computed. Afterwards the object pool is looped through starting from the offset position until an enabled object is found, which is then returned.

4.3 Sprite

Sprite is a class used to display display textures on the screen. They are generated from the engine and store pointers to the engines renderer and the texture which is to be displayed. Other properties held by the sprite class are a hide flag, indicating if the sprite should be drawn or not, a clip flag, indicating if the sprite displays the whole texture or only a part of it, the source rectangle, which is to be displayed from the texture (in case the clip flag is true), and a flip state, which determines if and how the displayed sprite is being flipped.
Displaying the sprite is implemented in the draw(...) method, which draws from the source rectangle on the texture to the destination rectangle on the screen. This method takes two integer values, where on the screen the sprite should be drawn. First it is checked if the sprite is hidden, if so the method returns immediately. Otherwise the destination rectangle is created using the position, which is given as parameters, and the height and width of the source rectangle, which is either stored as property or retrieved from the texture if the texture is not clipped. Then the sprite is rendered to the screen using the renderer, the texture, the source rectangle, the destination rectangle and the flip state.

```
1 void Sprite::draw(int x, int y)
2 {
3     if (hidden_) {
4         return;
```

```
    }
    SDL_Rect rect;

    rect.x = x;
    rect.y = y;

    if (clipped_) {
        rect.w = srcRect_.w * 2;
        rect.h = srcRect_.h * 2;
    }
    else {
        SDL_QueryTexture(texture, NULL, NULL, &(rect.w), &(rect.h))
    ; // retrieve width and height from texture
    }

    //Render texture to screen
    SDL_RenderCopyEx(renderer, texture, clipped_ ? &srcRect_ : NULL
    , &rect, 0, NULL, flip_);
}
```

4.4 Asset Manager

The asset manager is a class used to load and access texture assets. This class is never accessed from the game specific code, as the generic engine code should take care of handling the resources. Whenever a sprite is generated from the engine, the according texture is accessed using the asset manager, which loads a texture once and stores its reference to return it for future requests. This ensures to avoid duplicate loading of the same texture.
In Joust all sprites are generated from one big texture containing all sprites, so it is important to only load it once, as loading the complete sprite sheet multiple times would take up a lot of resources unnecessarily.
Internally the loaded textures are stored in a std::unordered_map [4]. A std::unordered_map is a container storing key-value pairs, using unique keys, which allows searching, inserting and removing elements an average constant-time complexity.

```
SDL_Texture* LoadTexture(const char* path, SDL_Renderer * renderer)
    ;
```

Accessing a texture via the asset manager is done using the loadTexture(...) method. This method takes the relative path to the texture as a constant char array as input. This constant char array is used as the unique key. First a look up in the std::unordered_map is done, looking if the key is already existing. If this is the case the according value, which contains a pointer to the texture is returned. Otherwise the texture is loaded and its reference stored in the std::unordered_map before returning it.

4.5 Component

Components [7, Chapter 14] are used to implement functionalities of game objects. The component class is a purely virtual class, providing an interface for implementing the derived classes. The declared functions concern the creation,

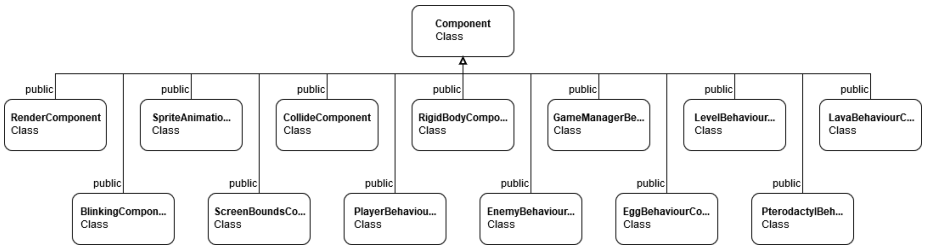

Figure 2: A class diagram giving an overview of the component class and all of its derived classes.

the initialization, the start-up, the update and the destruction of the component as well as receiving messages from other components. A component is provided with two pointers when it is created. These pointers are a pointer to the engine, which is used for example to read keyboard input or to create sprites, and a pointer to the game object, which contains this component, which is for example used to access the game objects position.

4.5.1 Render Component

The render component is used to visualize game object and is therefore part of all visible objects. To visualize the objects sprites as described in section 4.3 are utilized. When creating the component the sprite is loaded from a texture, which is in the case of Joust a PNG file. During the update, the component visualizes the sprite accessing the game objects position and using the sprite's draw() method.

4.5.2 Sprite Animation Component

The sprite animator component is used to animate sprites by changing the sprite's source rectangle at a variable rate. When creating the component it stores a pointer to a sprite visualized by a render component and an integer determining the amount of milliseconds each frame is shown in the animations. Afterwards the animations are added to the component by either giving an array of rectangles or defining the position of the starting frame, the width and height used by all frames and an offset between the rectangles. Additional information given when adding an animation is the amount of frames the animation contains and a flag indicating if the animation is looped or not. The animations are then stored in a std::unordered_map using a constant char array as key with the value being a data structure which contains all information relevant to the sprite animation.

```
1 struct Animation {
2     int frame_count;
3     bool looping;
4     SDL_Rect* frames;
```

```
5
6       Animation(int frame_count_, SDL_Rect* frames_, bool looping_ =
        false) : frame_count(frame_count_), frames(frames_), looping(
        looping_) {}
7   };
```

The currently played animation can then be changed by calling the playAnimation(...) function with the constant char array used as key as a parameter. In the update it is checked if the elapsed milliseconds since the frame of the animation was last changed exceeds the set milliseconds per frame. If that is the case the sprite is updated to show the next frame in the animation. If the previously displayed frame was the last frame of the animation it is checked if the animation is played in loop or not, if so the first frame of the animation is displayed, otherwise a message that the animation has finished playing is send to the game object and the sprite is not updated until a new animation starts playing.

4.5.3 Collide Component

The collide component implements testing for collision between its game object and a set of objects contained in an object pool, which is provided to the component on creation. For testing for collisions the component utilizes axis-aligned bounding boxes (AABB), stored as two points: the minimum (upper left corner) and the maximum (lower right corner) relative to the game object's position. In the update the AABB of the game object is tested against the AABBs of all game objects contained in the object pool. For each collision taking place the collision normal and the penetration depth are calculated and three messages are send: one collision message is send to the game object containing a pointer to the other game object from the object pool, the collision normal and the penetration depth and two hit messages one send to each game object involved in the collision, containing a pointer to the other.

4.5.4 Rigid Body Component

The rigid body component is used to apply forces on the game object it belongs to and changing the object's position accordingly. In the update the objects position is changed considering forces, acceleration and velocity, which are integrated using the first-order approximation Euler's method [1].

```
1 last_position_ = go_->getPosition();
2 go_->setPosition(last_position_ + (velocity_ * dt));
3 velocity_ = velocity_ + acceleration_ * dt;
```

When receiving a collision message the collision is resolved by calculating the new velocities for both game objects similar to the sphere collision in lab5 and setting their positions to the positions they had in the previous frame. Although the collide components are represented by AABBs the forces are calculated as if the objects were spheres, as most game objects in Joust using this component are not box shaped and handling them as spheres allows for a better approximation

than boxes. The addForce(...) method adds a force to the rigid body component, which will be considered when calculating the acceleration in the next update call.

4.6 Game Object

Entities in the game are implemented by game objects. Here the game object serves as a container object for various components implementing the functionality and as a state shared between those components. This shared state consists of a position, an enabled flag, an ID unique to each game object and a tag, allowing to label game objects as a certain type of object. The pointers to the components are stored in a std::vector as the amount of components is not known from the start and can vary from object to object.

```
bool enabled_;
uint32_t id_;
std::string tag_;
Vector2 position_;
std::vector<Component *> components_;
```

During the initialization, start, update and destruction method the game object loops over all components and calls the according method for the components. Furthermore the game object can forward messages from one component to all of the components contained in the game object.

4.7 Player Knight

The player knight is a game object consisting of seven components:

1. a render component for visualization of the knight.
2. a sprite animation component for playing the different animations of the knight, such as standing, walking, flying, etc.
3. a collide component for detecting collisions with enemy knights, eggs and the pterodactyl.
4. a rigid body component for applying the physics simulation for example gravity and resolving collisions.
5. a component defining the behaviour of the knight, implemented in PlayerBehaviourComponent: it handles user input, applies forces on the rigid body component to move the object, triggers playing the correct animations in the sprite animator component, keeps track of the player's lives and score and handles hits with eggs, enemy knights and the lava pit.
6. a component implemented in BlinkingComponent hiding and showing the sprite in 70 millisecond intervals generating a blinking effect, used to indicate invincibility.
7. a component implemented in ScreenBoundsComponent ensuring that the game object is always positioned within the bounds of the screen.

4.8 Enemy Knights

The enemy knights are game objects consisting of six components each:

1. a render component for visualization of the knight.
2. a sprite animation component for playing the different animations of the knight, such as standing, walking, flying, etc.
3. a collide component allowing other objects to detect collisions with the enemy knight.
4. a rigid body component for applying the physics simulation for example gravity and resolving collisions.
5. a behaviour component implemented in EnemyBehaviourComponent: it implements the enemy knight's AI, applies forces on the rigid body component to move the object, triggers playing the correct animations in the sprite animator component, handling hits with the player knight and the lava pit and spawns an egg when the knight is defeated
6. a component implemented in ScreenBoundsComponent ensuring that the game object is always positioned within the bounds of the screen.

4.9 Eggs

The eggs are game objects consisting of five components each:

1. a render component for visualization of the egg.
2. a collide component allowing other objects to detect collisions with the egg.
3. a rigid body component for applying the physics simulation for example gravity and resolving collisions.
4. a component defining the behaviour of the egg, implemented in EggBehaviourComponent: it restricts movement of the object, by damping the forces applied on the rigid body component, ensuring the egg to stop after traveling a certain distance, disables the egg when hitting the player knight or lava and respawns an enemy knight after ten seconds of not being disabled.
5. a component implemented in ScreenBoundsComponent ensuring that the game object is always positioned within the bounds of the screen.

4.10 Pterodactyl

The pterodactyl is a game object containing six components:

1. a render component for visualization of the pterodactyl.
2. a sprite animation component for playing the moving and death animations of the pterodactyl.
3. a collide component allowing other objects to detect collisions with the pterodactyl.
4. a rigid body component for applying the physics simulation for example gravity and resolving collisions.
5. a behaviour component implemented in PterodactylBehaviourComponent: it implements the pterodactyl's AI (constantly following the player knight), keeps the velocity on the rigid body at a constant magnitude, triggers playing the correct animations in the sprite animator component, stuns the pterodactyl (not following the player knight) for 1.25 seconds after collisions with the player and awards the player with a score when the pterodactyl is defeated.
6. a component implemented in ScreenBoundsComponent ensuring that the game object is always positioned within the bounds of the screen.

4.11 Arena

The arena consist of two game objects: the arena platforms and the lava, containing ten and three components respectively.
The arena platforms consist of one render component for visualization, a behaviour defining component implemented in LevelBehaviourComponent, which resolves collisions in place of a rigid body component and a collide component for each of the eight platforms for detecting collisions with the player and enemy knights, the pterodactyl and the eggs.
The lava pit consists of a render component for visualization, a behaviour component implemented in LavaBehaviourComponent, which resolves collisions in place of a rigid body component and a collide component for detecting collisions with the player and enemy knights and the eggs.

4.12 Game Manager

The game manager is a game object containing only one component: the GameManagerBehaviour. This is responsible for spawning a new wave of enemies once all enemies are defeated and eggs collected, displaying the players lives and score on the UI and displaying a game over message when the player has lost all lives.

5 Conclusion

While the architecture is sufficient for the implementation of Joust, it does have its shortcomings and could be improved upon in some parts, which is discussed here.
The main purpose of the game objects in this architecture is to function as container objects for the components. While the message system between components of one game object generally allows decoupling these components, this decoupling is broken at times, e.g. in the PlayerBehaviourComponent, where multiple accesses to the rigid body component and the sprite animation component are required every frame and doing this in messages would flood all components with unneeded messages, while also increasing the line count of the class and possibly worsen readability. Decoupling is also broken at some points due to the lack of a proper message system across game objects, which is e.g. the case in the GameManagerBehaviour class. This could be improved by implementing the observer pattern [7, Chapter 4]. The components could have also been stored in arrays of each component type. During the game loop the components belonging to the same array could have been updated collectively and this way improving data locality by implementing locality of reference. Data locality [7, Chapter 17] has not been a problem during the development of Joust, which is why the engine architecture was not redesigned to improve it. Further improvements concern the collision detection which could be implementing a space partitioning tree [7, Chapter 20] to improve performance by reducing the amount of game objects iterated over every frame. Another possible improvement could be using a second-order method such as leapfrog [3] or Verlet integration [6] to reduce the error compared to the first-order method currently used to approximate the changing velocity and position in the rigid body component.

References

[1] Euler method - wikipedia. https://en.wikipedia.org/wiki/Euler_method. Accessed: 2019-03-16.

[2] Joust (video game) - wikipedia. https://en.wikipedia.org/wiki/Joust_(video_game). Accessed: 2019-03-16.

[3] Leapfrog integration - wikipedia. https://en.wikipedia.org/wiki/Leapfrog_integration. Accessed: 2019-03-16.

[4] std::unordered_map - cppreference.com. https://en.cppreference.com/w/cpp/container/unordered_map. Accessed: 2019-03-16.

[5] std::vector - cppreference.com. https://en.cppreference.com/w/cpp/container/vector. Accessed: 2019-03-16.

[6] Verlet integration - wikipedia.
https://en.wikipedia.org/wiki/Verlet_integration. Accessed: 2019-03-16.

[7] Nystrom, R. Game programming patterns.
http://www.gameprogrammingpatterns.com/contents.html, 2014. Online.
Accessed: 2019-03-16.