Game programming (KGGAPRO1KU) exam project

Thomas Hoffmann Kilbak (thhk@itu.dk)
Kristoffer Bruun Højelse (krbh@itu.dk)

IT University of Copenhagen - January 3, 2024

1 Introduction

In this project, we will implement our own version of the game **PlateUp!**,[1] which we call **!PlateUp** (pronounced "Not PlateUp").

The game is about you owning a restaurant, where you have to serve your customers what they order, before they run out of patience. It is up to the player to optimize their restaurant, by moving around tables to get the most efficient layout. The game runs through a number of "days". Each day there will be a certain number of orders to complete. Once you have completed the day, you will be able to edit your restaurant, to make it as easy as possible for you to serve your customers in time. As the game progresses, there will spawn a new dinner table that customers can order from, which you will have to accommodate in your restaurant. In addition to new tables spawning, each day will be harder than the previous due to these factors:

- Customers will wait shorter amounts of time for their order.
- There will be more orders to complete during the day.
- The time between each order being placed will be shorter and shorter.

You can lose in one of two ways. 1: A customer waited too long for you to serve their meal. 2: A new order should be placed, but you have no available tables.

Figure 1 shows a screenshot of the game. The white tables indicate tables that customers can order from. In the figure, there are two current orders, one for a carrot, and one for a tomato. The indicator for the tomato is slimmer, which means that there is less time to deliver it. In the figure, the player is currently carrying a tomato, which has been picked up from the box with a tomato on it.

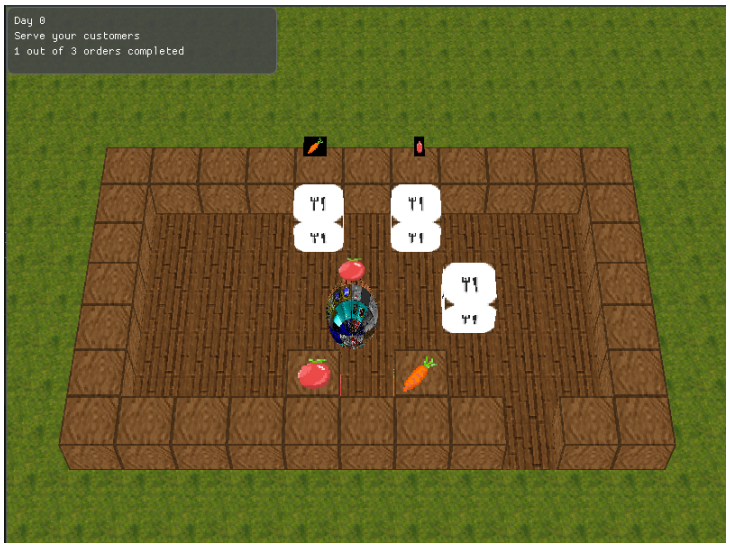

Figure 1: Screenshot of the game.

[1] `https://www.yogscast.games/plateup`

1.1 Focus of the game

We have decided to focus on implementing as much of the gameplay of PlateUp! as possible. This means that we need the basic systems of e.g. customers ordering, collisions, and moving tables to be in place, before we can implement more advanced features such as combining ingredients to make dishes. We want to implement a "vertical slice", meaning that we want to implement a broad set of all game elements, without any single part of the game necessarily being super polished. "Nice-to-haves" such as music and sound effects were therefore not prioritized. Textures were also not prioritized, with us focusing on making them clearly distinct, but not very pretty.

We decided to make the player movement rather stiff without turning-animations, as the direction of the player is important for interacting with boxes, and making a smooth way of turning that did not feel clunky was difficult. Additionally, we decided to not animate customers walking into the restaurant, as making the animation would be quite difficult, and affect our ability to implement gameplay features.

2 Architecture and design

The code is based on Kristoffer's submission of the Wolfenstein exercise. This includes ComponentRendererMesh, which we have repurposed as ComponentRendererSquare, to render boxes and floor tiles. Additionally, we use ComponentFollowTarget from the Flappy bird exercise, which is used to make the picked up items follow the player's position.

2.1 Defining the level

We have placed as much of the configuration of the game as possible in the scene.json file. In addition to defining the game objects to create, it also defines the play-area. This includes where boxes, emitters, and consumers should be initially placed, as well as where there is floor, and which texture should be used for that given floor tile. This means that we potentially can define many different restaurants, with different shapes, boxes and types of floor.

2.2 Classes

The overall design is GameObject-Component architecture, as this is the design pattern used for SimpleRenderEngine.

An important part of the game, is being able to interact with tables in order to pick up ingredients, serve food, and place items where they are wanted. We have made a class **ComponentInteractable** which inherits from Component for this purpose. This class only defines an ID which is the type of item that it produces or receives, as well as an overridable function Interact.

Three classes inherit from ComponentInteractable, those being **ComponentConsumer**, **ComponentEmitter** and **ComponentTable**. A consumer is a thing that can consume an item from the player, in our case, a consumer is a customer with an order. An emitter is a thing that can emit an emit to the player, in our case, that is the boxes with food. Tables can be used to put items down, such that they are easily available if needed in the future.

Using ComponentInteractable means that any game object can have an interactable component, such that we can define what happens on interaction in each class, rather than having to decide in the Controller. This gives the code lower coupling. This can be seen on figure 2, where the controller does not depend on Emitter, Consumer or Table, but only their parent class, Interactable.

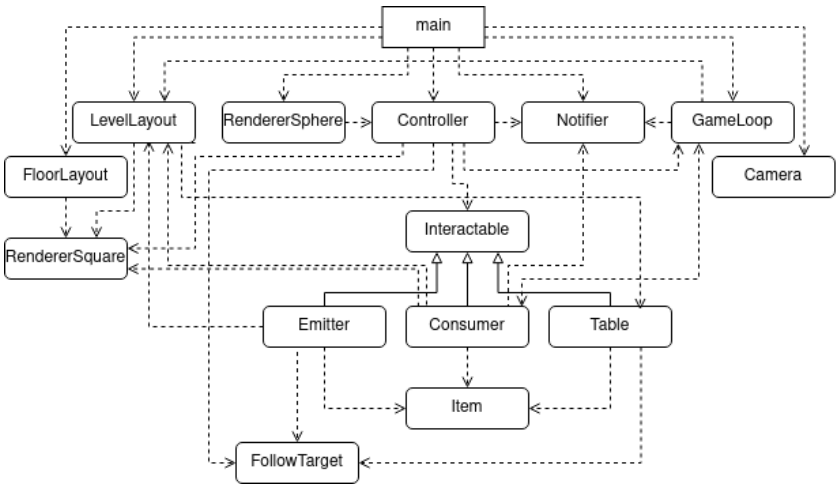

Figure 2: Class diagram of the components of the game. Main is not a class, but is included to show which components are initiated by main. In code, the names of all classes are prefixed with "Component", which is not shown here for simplicity. All classes inherit from MyEngine::Component, which is not shown.

On the class diagram, all of the dependency arrows might give the impression that the code has very high coupling. It might very well be possible to reduce coupling, but we believe most dependencies make sense. For example, FloorLayout, LevelLayout, and Consumer depend on RendererSquare, as FloorLayout and LevelLayout use it to create the level, and the consumer uses it to make the indicator for which food the consumer has ordered. Thus, we have attempted to keep components as self-contained as possible and avoid excessive dependencies.

2.3 Coordinates

We utilize a 2D coordinate system, to keep track of where items are located. This is done by adding game objects with names that include the location. For example, a box located at x:3 and y:4 would be a game object with the name "box-3-4". The operation of renaming is a lot less expensive than destroying and creating the object anew.

```
bool GameObjectExists(std::string name) {
    return _gameObjects.find(name) != _gameObjects.end();
}

void RenameGameObject(std::string old_name, std::string new_name) {
    if (!GameObjectExists(old_name)) return;
    auto go = _gameObjects[old_name];
    _gameObjects.erase(old_name);
    _gameObjects[new_name] = go;
    go->SetName(new_name);
}
```

Figure 3: The functions that we have added to the engine to allow moving boxes and checking for the existence.

Because the engine uses a map to keep track of game objects, we can look up objects in constant time. Thus, when the interact-button is clicked, we get the coordinate that the player is looking at, and check if that coordinate has a box. If the game objects were on non-integer coordinates, multiple objects on the same coordinate or not 1x1-sized objects, it would have not made sense to do it this way, but in our case, this architecture makes a lot of sense.

Whenever the player picks up a box or an item, the game object is renamed to "box-held" or

"itm-held" accordingly, and when it is placed, it is named according to its new location. This helps inforcing that you can only hold one object at a time.

2.4 Collisions and physics

Collision detection is important, as the player should not be able to walk through boxes. To do this, we are using the ComponentPhysicsBody of the provided engine, which uses Box2D under the hood. To use this for our purposes, we needed to change the coordinate system, such that the floor is in the x-y-plane rather than the x-z-plane.

2D collisions are sufficient in this case despite the game using 3D graphics, as the player only moves in two dimensions. The boxes each have a square PhysicsBody, while the player has a circular one. PhysicsBody is also used to apply linear velocity for movement, and impulse when dashing.

2.5 Sprites

We made our own sprites in Photoshop, each 32x32 pixels. The textures were packed into a single sprite sheet using TexturePacker. We implemented our own method, such that we can create a square with a named texture, rather than having to specify uv's for all of them. The way we have done it, is we look up the named texture in the sprites.json file, and the uv's are specified based on the metadata from the JSON.

For the player, we had a hard time getting a texture to wrap correctly around the oblong sphere. Thus, we decided to use the sprite sheet from the Wolfenstein exercise. This does not make the player look like a chef like we wanted, but it does clearly show when the player changes direction, which is important, as interactions always occur in front of the player.

3 Performance evaluation

Proving that a program does not have resource leaks, is a close to impossible task, but we can reason about the mitigations we have made. The use of smart pointers throughout the code, should mean that a resource leak is less likely, as a shared pointer that has no references to it, will be automatically freed.

To test for any major memory leaks, we tried to run the game through its various phases, and observed the memory usage, with the following results.

- Game start: 36.0 MB
- After moving around tables: 36.0 MB
- Playing day 0: 36.1 MB
- Playing day 5: 36.1 MB
- Playing day 10: 36.2 MB
- After losing at day 10: 36.2 MB

It can be seen, that despite spawning and deleting indicators and running the game loop many times, the memory consumption only increases slightly. This makes us believe, that there are no noteworthy memory leaks, where variables are not deleted from memory. One oddity, is that the memory consumption does not fall after losing. We would expect it to fall slightly, as tables spawned during the progression of the previous game are removed. That said, the memory consumption is only 200 kB higher than expected, which is minuscule, so this may be due to other factors such as the operating system waiting to release resources.

It appears that the SimpleRenderEngine uses either a fluid or fixed game loop with a target fps of 60. On our computers, we never exceed the 16.7 ms frame time, so we believe that there are no major performance issues with the code, and therefore no hard bottlenecks. That said, there are possible performance improvements, which we discuss in section 4.2.

To benchmark the CPU and GPU utilization, we try to run the game on a laptop with a 16 thread Ryzen 7 4800U[2] CPU. The game utilizes about 3% of the CPU, meaning about 48% of a

[2] https://www.amd.com/en/support/apu/amd-ryzen-processors/amd-ryzen-7-mobile-processors-radeon-graphics/amd-ryzen-7-4800u

single thread, while the GPU utilization is about 12%. Thus, we still have lots of room to add additional computations, before the frame rate of the game will be affected. As the CPU has the highest utilization, that would probably become the bottleneck before the GPU. Thus, if the game is further expanded, care should be taken to not add many expensive computations to e.g. the game loop, which will further strain the CPU.

4 Result

We consider the resulting game a good baseline version of PlateUp!. Many features are missing from the original, which means that our version is lacking in entertainment. However, it has the fundamental game loop necessary, many of the important mechanics of the game are present (such as picking up items and moving around tables), and the code has an architecture that can easily be expanded upon. We have several ideas for how to improve the game.

4.1 Possible gameplay improvements

Gameplay-wise, there are several features from the original PlateUp!, which have not been implemented. Most notable, is the ability to combine ingredients to create more advanced meals. We did implement the ability to place ingredients on tables, but this is currently rather pointless, but would be very useful, if e.g. placing a carrot and a tomato together, would create a salad.

We would also have liked to add more types of foods. This can be easily done, as all that is needed is adding sprites for the food, as well as the emitter that provides the food. We would have liked to introduce more types of food gradually, the further the player progressed through the game, to increase difficulty.

If these features were added, another great addition would be multiplayer. Working together to serve dishes, is one of the greatest things about the original PlateUp!. Adding this should be doable by adding another controller that accepts other keyboard inputs. Some modifications would have to be made elsewhere too. For example, we currently assume that there is only a single held item at a time. In the case of multiplayer, both players should be able to hold items simultaneously.

Making customers walk into the restaurant by following some shortest path produced by e.g. Dijkstra's algorithm, would be a cool feature, but a nice to have feature, since it's mostly a visual indicator rather than a game mechanic. However, emphasizing the fantasy of customers arriving at your restaurant would make the game more immersive and engaging.

Resetting the game after losing, should preferably be done by deleting most game objects, and calling the Init function on ComponentGameLayout again.

4.2 Possible performance improvements

We could avoid creating and rendering a large portion of the squares, as the camera is in a fixed location. This means that the sides of boxes facing away from the camera could be omitted, as well as the grass that is not in frame. Also, because the boxes often will stand next to each other, the sides that are next to each other, do not need to be rendered either.

Currently, the default stage includes 625 floor tiles. This could be reduced to about half, if we avoided rendering the ones outside frame. We decided not to try to prune elements as, 1: We did not encounter performance issues yet. 2: Whatever algorithm we would use to decide which faces to prune, could be prone to introducing bugs where some face might not be rendered by mistake. 3: We would like to keep opportunities open to move the camera if we wanted, and allow for the player to move around the boxes as desired. Additionally, it appears that SRE implements some face culling, though we did not look further into this.

Furthermore, we fetch the metadata for the sprite sheet whenever a RendererSquare is initiated, which happens hundreds of times at startup. This likely makes the game slower to start, but the game still starts in about 2 seconds, which is why we have not changed it. Alternatively, we could load the metadata in LevelLayout or FloorLayout, but this would increase coupling, as RendererSquare would become dependent on one of them.

5 Responsibilities

The game was made primarily using pair-programming, with alternating roles of who is the driver and who is the navigator. Thus, most parts of the system have been worked on by both of us. A few parts of the segment were primarily worked on by one person though.

The movement of the player is based on Kristoffer's Wolfenstein exercise, thus he implemented that. Kristoffer also implemented most of the collisions for the boxes and the player.

Thomas made the initial game loop.

Refer to the git history to see who have committed what.[3] Note that for most of the history, there are a few commits by one person, then a few by the other. This indicates that we swapped driver/navigator roles.

[3] `https://github.itu.dk/krbh/notplateup`