Tron(1982)

Chi Hong Chao
Game Engine Architecture, TDA572 / DIT455
A.A. 2020/21

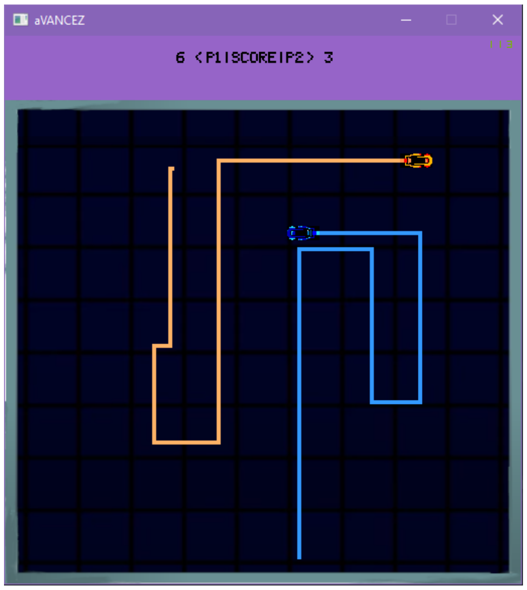

Figure 1: A picture of the game. *2 Players* are attempting to force each other into the walls or the edge in order to win.

1 Introduction

This report describes the implementation of the light cycle mini game from the 2D game **Tron**, using an architecture according to the principles depicted in [3]. The corresponding source code is in the solution of GEATron. The

original **Tron** was created and published in 1982 and was extremely popular at the time, leading to some people calling Snake games ”Light Cycle games”[2]. The original mini-game features a single player moving a futuristic motorcycle around the screen in a grid-like behaviour. This motorcycle, called a light cycle within the Tron universe, creates walls of light where any light cycle can crash into. The player is also able to slow down the speed of the bike. The objective of the game is to force one or more enemy cycles into crashing into these trails or into the edges of the screen while avoiding them yourself at the same time(See Fig. 1). In the original, the player would play through the 4 mini games in increasing difficulty endlessly. For the light cycle mini-game, a higher difficulty meant more enemies.
In this report, this light cycle mini-game will simply be referred to as ”Tron” unless specified.

2 Specification

Several modifications were made in contrast to the original Tron. Instead of a single player against computer-controlled enemies, a 2 player solution was implemented, scrapping the increasing difficulty mechanic. Furthermore, this implementation adapts the scoring system to 2 players, and does not high score that is saved across instances of the game. Finally, there is no difficulty progression when a player wins. The objective, however, stays the same - defeat the enemy by forcing them into walls or the edge.
Tron consists of two basic entities: the players' light cycles and the wall trails produced. The goal of the game is to force enemies to crash first.
The Player Light Cycles. The player light cycles are positioned on opposite sides of the screen. They automatically move forward, but their speed can be adjusted. In this implementation, players can change between 3 different speeds - slow, medium, and fast - at any time. The player can turn 90 degrees left or right relative to their current direction. The bikes do not have lives - they explode immediately after hitting a wall or edge of screen.
The Walls. Walls are produced behind the light cycles as the cycles move forward, forming a ”trail” behind it. The walls are continuous such that no gaps are created in between. They are also indestructible. The walls do not discriminate - any light cycle explodes upon hitting any wall. Since a trail is created, it could be said that the walls also illustrate the path that each light cycle has taken within the game.

End State. The last surviving light cycle wins. Since this is a 2 player version, if Player 1 crashes, Player 2 wins, and vice versa.
Additional Feature - Background. The background can be changed during runtime.

3 Architecture Overview

In **Tron**, there are two game entities - light cycles and walls - interacting with each other. The architecture, however, contains additional objects which facilitate this interaction.
Game entities have a collection of components with a minimal state, shared among all the components. The components implement the functionality of the game object. The game object also implements the communication mechanism through messages, which are used to notify other game objects of events that are of interest. There are three basic kinds of components: Behaviour, Render and Collision. Behaviour defines the logic of the game object, and in Tron's case, primarily how it reacts to the input of the user and the state of the game(game over). Since it was modified to be a two player game, there is no artificial intelligence in the logic. The render component visualizes the game entity through the placement of sprites according to the state of the game object, and the collision component is responsible for detecting if the game object is colliding with other entities.
In addition to game entities, a singleton-like class is used to keep track of the state of the walls in the game, and another to render/visualize the walls. They are singletons as there is only one instance of these two classes at any time, but are not singletons such that they are not globally accessible nor have typical singleton methods (getInstance(), for example). These classes are not game objects, and thus do not have positions and messaging abilities. Since the class that keeps track of the walls is a uniform grid, it also facilitates the collision detection between walls and players.
All the entities and classes are handled by the game class. This class is created at the entry point of the application in the main procedure, and then it is updated in the main loop. The game class carries out three main tasks during the update:

1. update all the enabled game entities in the game;
2. update the singleton-like entities;

3. verify the winning and losing conditions, and react accordingly.

The game class is also responsible in visualizing the interface and background of the game, which essentially has the overall state of the game. The game state knows the score and if the game is over. Furthermore, the game class plays a relatively large role in sound. In an ideal architecture, the background, interface, and sound should all be in separate objects, and the game class simply updates these entities as well. However, there was little time and refactoring these things were not prioritized. It could be argued that these aspects of the game are quite small and thus could be done by the game class.

4 Implementation

4.1 Game Update

The game update loop is implemented as described in [3] Chap. 9 and 10. The entry point `main(...)` is in `tron.cpp`. The `game` class is then created and then updated, which then creates and updates the game objects. When the user hits the `'q'` or the `ESC` keys, the loop exits and the data structures are de-allocated. If the user inputs `'r'`, the game is reset to the starting state. During the update, the time `dt` elapsed for computing the last frame is measured and indirectly used to update the player light cycles. This ensures that the motion on slow machines is the same than on faster ones.

4.2 The Player Timer

Here, the timer is said to be indirectly used as `dt` is added to a timer in player. Only when the timer reaches a certain value is the player allowed to update, shown below:

```
timer += dt;
const float TIMER_CD = 0.05f;
if (timer > TIMER_CD) { {...}
        if (currentDirection == 0)
        { go->verticalPosition += -speed; }
        if (currentDirection == 1)
        { go->horizontalPosition += speed; }
    {...}
}
```

This method was used because it was difficult to snap the player to the grid by multiplying speed with dt. For example, moving 4 pixels each update could not be easily guaranteed. As such, a timer was used, and when a certain threshold was reached, the player would move exactly 4 pixels, thus conforming to a "grid". It should be noted that this timer was implemented before the uniform grid, described below, was finalized, leading to the movement not depending on the grid at all to function. Since this timer is only linked to the player, the game's Frames per Second (fps) is not affected.
A few issues are created by this implementation. Firstly, the timer functions as a Frames per Second (fps) limiter for the player, where the threshold number determines how fast the player updates. While computers with high fps are not affected, those with lower fps have lowered player speeds and sluggish input recognition. Secondly, since player input is not accepted before the timer threshold is reached, the responsiveness of player input is heavily dependent on the threshold, where a lower threshold means higher responsiveness.
Since the threshold is tied to the fps and player input, simply lowering the timer threshold will make the player move too quickly. One solution is to make the grid cells smaller, but this would greatly affect all of the other objects, generating new problems. As such, due to the lack of time and unknown variables, this sub-optimal solution was settled on. In hindsight, the timer could be omitted by having the player snap to the uniform grid, which could increase responsiveness while removing possible bugs.

4.3 The Uniform Grid

The Uniform grid is a spatial partitioning pattern that helps facilitate locating objects ([3] Chap. 20). Here, the uniform grid is specifically used to create, locate, and remember where each wall is and who it belongs to. As such, instead of creating a "wall" game object for each cell, the grid keeps track of the state of walls. This makes it much more efficient and flexible compared to having wall objects placed in each cell. Other objects, such as the player's collision component, utilizes this information to see if the player is colliding with any wall. It is important to note that the Uniform Grid does not visualize the walls - only the state it saved, allowing the visualization of the walls to be completely decoupled from the logic.
The uniform grid's Update function has two functions:

```
void UniGrid::Update(double dt) {
        UpdateState(dt);
        CheckCollisions();
        {...// hash table stuff but unused here}
```

4.3.1 UpdateState()

UpdateState contains the majority of the grid code, as it goes through the entire grid determining what type of wall each occupied cell should contain. In order to do so, UpdateState needs to go through each player light cycle in the game, then state of the current light cycle is obtained. More specifically, the grid needs to know the current speed, in order to know where to start placing walls, and how many walls to place behind the bike. An offset, called positionWidth, tells the grid where to start placing walls. The basic idea is that if speed = 0, place wall behind 1 grid cell. If speed = 1, place walls behind 2 cells.
The current direction of the light cycle is also needed to know which way to start placing these walls. As such, the grid uses both the direction and speed information to correctly place walls. The code here has been simplified to conserve space.

```
int minPosX = currentCycle->horizontalPosition /cellSize;
int minPosY = currentCycle->verticalPosition /cellSize;
{... etc}
// determine cells to place walls
if (currentCycle->GetComp<BehavComp*>()->currDirection()==0){
  minPosY =(currentCycle->vertPos - positionWidth)/cellSize;}
if (currentCycle->GetComp<BehavComp*>()->currDirection()==1){
  maxPosX =(currentCycle->horizPos + positionWidth)/cellSize;
} {// repeat for other directions...}
```

At this point, the grid knows which cells to add walls. One large issue I had was how to determine what type of wall is placed at that specific cell in order to connect the walls properly. First, I settled on a DirectionState enum for better readability. Each grid cell is an object which contains a pair¡bikeID, wallState¿ that determines the state. To know which specific light cycle it is, I simply add 1 to the bike's vector index. This method clearly is not the best, but i found it to be the most simple - the vector of lightcycles in game.h never should be changed in the first place anyways, so the indexes of the bikes should always be the same.

```
////// GameObject.h //////
```

```
enum DirectionState { EMPTY = 0,  VERTICAL = 1, HORIZONTAL =
    2, TOPLEFT = 3, TOPRIGHT = 4, BOTRIGHT = 5, BOTLEFT = 6};
////// UnigridCell.h //////
std::pair<int, int> state; // id= index+1, wallstate
```

The states are then placed into each cell. Once again, the current direction is needed to know what type of wall to place.

```
for (int a = minPosX; a <= maxPosX; a++) {
          for (int b = minPosY; b <= maxPosY; b++) {
          uniGrid->grid[a][b].state.first = i + 1; // id
          if (currentCycle->GetComponent<BehaviourComp*>()->
              getCurrentDirection() == 0) {
                    uniGrid->grid[a][b].state.second = VERTICAL;
```

Originally, this was straightforward - up/down meant vertical walls, and right/left meant horizontal ones. However, the wall "connections" were never robust, and depending on when the player shifted directions, walls would sometimes jut out. Corner wall states then came into being. To know where to place corner walls, the grid also retrieves all the possible previous directions of the bike relative to the current direction. For example, if the bike is currently going up, it could only have come from either right of left:

```
// big brain moment
// only check corners for the closest cell behind lightcycle
if (a == maxPosX && b == maxPosY) {
  if (currentCycle->GetComponent<BehaviourComp*>()->
      getPrevDirection() == 1) {
            uniGrid->grid[a][b].state.second = BOTRIGHT; }
  if (currentCycle->GetComponent<BehaviourComp*>()->
      getPrevDirection() == 3) {
            uniGrid->grid[a][b].state.second = BOTLEFT; }
}         {...// repeat for all other directions}
```

4.3.2 CheckCollisions()

CheckCollisions is much simpler, but the name is now misleading after a few iterations of changes. Now, it simply removes the walls of a particular bike if it has crashed. This function was what led the state to be changed into a pair rather than a simple int. This makes it much easier to add new bikes.

```
//remove walls of that bike
if (currentCycle->enabled == false) {
for (int a = 0; a < this->grid.size(); a++) {
for (int b = 0; b < this->grid[a].size(); b++) {
```

```
if (this->grid[a][b].state.first == i+1 && this->grid[a][b].
   state.second != EMPTY ) {
  this->grid[a][b].state.first = EMPTY;
  this->grid[a][b].state.second = EMPTY;
```

4.4 WallController

The WallController is also poorly named. After several modifications, the WallController now only has one responsibility: To draw the walls onto the screen. To do so, the drawWalls() function goes through each cell in the grid for each frame, and draws rectangles onto the cell based on the state (code simplified):

```
for (int i = 0; i < this->unigridRef->grid.size(); i++) {
for (int j = 0; j < this->unigridRef->grid[i].size(); j++) {
if (this->unigridRef->grid[i][j].state.second == VERTICAL){
 engine->fillRect(vertMinWallPos,vertMaxWallPos,p1r,p1g,p1b);
 // do the same for horizontal and corner walls
```

While the WallController specifies the coordinates to draw the rectangles, it calls the engine to actually draw the rectangle, separating concerns.

4.4.1 Singleton?

Unlike Gameobjects and Components, both the Uniform Grid and Wallcontroller only have one instance of the object during runtime. In this sense, it has the attributes of a singleton. However, Nystrom[3] in Chapter 6 describes a Singleton as **ensuring** having only an instance running, and providing **a global point of access to it**. Both the grid and wallcontroller do not have functions which ensure having an instance, nor are the instances globally accessible. Furthermore, there are no functions which return that specific instance for other objects to use.
Like Nystrom suggests later in the chapter, the unigrid instance is passed in as a parameter for objects and components which require it. The reason for setting up the Uniform Grid to take care of the entire screen's wall state was twofold - it was a simple concept, and it allowed for more efficient collision detection down the line too. This caused the Wallcontroller to also be a singleton. The idea of letting players draw their own walls was considered as an alternative. However, I deemed it to be more complicated than necessary - the entire grid would need to be passed through as many times as there are bikes, making it slower.

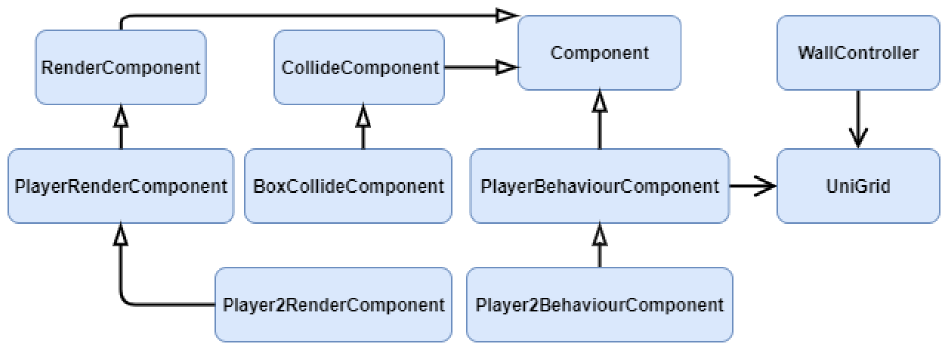

Figure 2: Class diagram for the component class and its derived classes, along with the relationship between the behaviour components, uniform grid, and wall controller. Here, the hollow arrow signifies "derived from", and the pointy arrow "uses".

4.5 Component

A component implements a particular functionality of a game object, as described in [3] Chap. 14. In Tron, the Player class is mainly the one which takes advantage of the component, since it is one of the few game objects. As such, much of the discussion below will be focused on the Player's components. The class implementing a component is *purely virtual* and it offers an interface to be implemented by the *derived* classes. The interface is concerned with the creation, the initialization, the update and the destruction of the component. Three pointers are provided in the creation of the component, respectively:

```
AvancezLib * system;
GameObject * go;
std::set<GameObject*> * game_objects;
```

where:

- `system` is used to access the engine; for example, it is used by the PlayerRenderComponent to animate, load and draw sprites, and it is used from PlayerBehaviorComponent of the Player and Game classes to read keyboard input;

- `go` is the game object this component is part of. This allows components to access properties of the game objects itself, or even other components of the game object if needed.

- `game_objects` is the global container of game objects. This is useful because the component may need to create objects on the fly and insert them in the global collection of entities. In Tron, however, this is not used extensively, but could be useful in the future if more features are added.

4.5.1 Render Component

The render component is used from all the visible objects in the game to visualize themselves. For the player, the visualization is carried out through the use of a vector of 2D sprites. During the creation of the component, the sprite is loaded from a bitmap image stored on the disk. Then, during the update, the component checks the state of the game object to which it belongs to, chooses the correct sprite, and visualizes the sprite in the specified position. It is important to note that this specified position can vary from the game object's position.

4.5.2 Collision Component

The collision component is used to test for collisions between the game object to which it belongs, and in the Player's case, a specific range of cells in the Uniform Grid. During the update function, the bounding box of the game object is tested against the bounding box of other objects. In the Player's case, it is tested against a range of possibly colliding grid cells. In case the collision occurs and one of the cells have a wall, then the collision component sends a `WALLCRASH` message to the two objects which will handle the response to the collision. Since the Uniform Grid is not a game object, the Player collision component specifically only sends this message to itself.

4.5.3 Behaviour Component

The behavior component contains the logic of the game objects, and is probably the messiest component in the Player. It handles the user controls in case of the player, using the timer concept explained above. It also reacts accordingly to the different states of the game object.

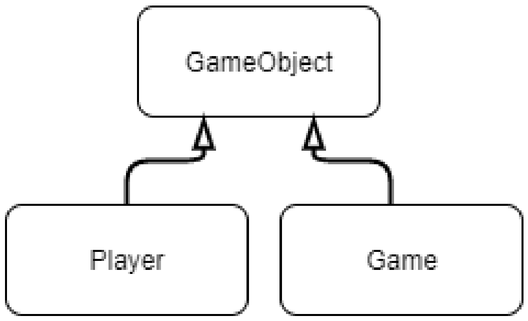

Figure 3: Class diagram for the game object and its derived classes.

4.6 Game Object

A game object implements a generic entity in the game. All the functionality is implemented in a separate set of components, and the purpose of the game object is defined as:

1. it stores the state which is common to all the objects;
2. it stores a collection of components which actually defines the object;
3. it handles the messages received by other objects.

A game object can store an indefinite number of components. The pointer to the components are stored in a `std::vector` because the number of components are not known until runtime. During the update, the game object iterates over all the components and updates each of them.

4.7 Player

The player is a game object composed by four components:

1. a behavior component implemented in `PlayerBehaviorComponent`, it handles the input concerning the specific player's light cycle. Specifically, it places the game object into the correct position according to input. Furthermore, it stops accepting input when it has crashed. There is a specific behaviour component for both Player 1 and 2, due to differing input keys. Player2BehaviourComponent inherits from Player-BehaviourComponent, so much of the code is reused.

Since there are so few sounds, the "motorcycle" audio initiates playback in this component as well, which is bad design. However, due to time constraints, this was left in. While the original idea was to have a distinct motor sound for both players, this was scrapped and a single, non-overlapping sound is played for both players.

2. a render component called PlayerRenderComponent for visualization. Again, Player 2 has a different component due to different sprites. The render component also deals with animating the explosion when a bike crashes.

3. a collision component called BoxCollideComponent, which queries the uniform grid to not only see whether the collider is hitting any cells with walls in the current frame, but also whether or not it has passed any cells with walls since the last frame. Furthermore, this collision component queries the WindowColliderComponent to check if the light cycle has hit the edges of the screen, also resulting in a *WALLCRASH* message.

4.8 Game

The `game` class is a container for all the game objects in the game. Strangely enough, it is also a game object itself, and I can only see bad unforeseen consequences from this. This could however also mean that multiple instances of game can be created, or that components in game can be created to deal with, for example, a sophisticated scoring system. In Tron, this `game` class also reacts to user input for changing backgrounds, as well as the playback of audio during the "game over" state.
In the creation phase, the Uniform Grid, WallControllers, and Game objects are created and initialized, preparing them to be able to update. The game objects' components are created as well, then passed to the corresponding objects. The objects are then inserted in the global collection of objects `game_objects`. A starting audio clip is also played.
During the update, `game` updates all the currently enabled game objects along with checking whether or not the game over state is activated. Since the game over state is triggered by a message, an immediate response is taken within function itself, but audio and explosions take a longer time, which need to be timed correctly, requiring a delayed response only possible

in the Update function. Furthermore, the game should still be able to accept certain inputs after the game is over, for example when the player presses "r" to reset. The collection of game objects is stored in an `std::set` [1]. A set is an associative container which avoids inserting objects more than once. So, a game object is not inserted if it is already in the set, avoiding unnecessary duplications.
Note that such set is initialized for storing generic objects:

```
std::set<GameObject*> game_objects;
```

During the game, derived objects are inserted into the set (the player, the window (which houses the window collider)), for example:

```
player = new Player();
...
game_objects.insert(player);
```

When the `update` method is called on the elements of `game_objects`, the polymorphism mechanism calls the corresponding method in the derived class.

4.8.1 Game Class Issues

Currently, the game class is most problematic of all classes as it handles many small but different responsibilities on top of simply updating game objects. This issue was simply due to time and prioritizing more pressing matters. One of these responsibilities is audio playback. Currently, audio playback code is strewn across in little bits and pieces across `game`:

```
void showGameWon() {
    int levelwonXpos = 180; int levelwonYpos = 37;
    engine->finishSound();
    if (winner == 1) {
      levelwonString = "=== PLAYER 1 WON! ===";
    if (scoreIncreased == false) {
      p1Score++;
      scoreIncreased = true;
    }
    if (crashplayed == false) {
      engine->PlayMp3(0, 1);
      crashplayed = true;
    }
    if (Mix_PlayingMusic() == 0) {
      if (erynoiceplayed == false) {
        engine->PlayMp3(0, 3);
        erynoiceplayed = true;
}}}
```

There are clear problems here. While the function is meant to draw text if the game is over, the crash audio playback is also initiated here. Although it could be argued that another function playGameWonAudio() could be made, the underlying issue is deeper - a separate Audio object should be made, which the game class should call. Another problem is how some audio functions are wrapped by the engine, but an SDL_mixer function is also directly used. While benign here, mixing these more later on will create many dependency and coupling issues at no advantage. Here, I simply ran into a bug which I couldn't fix unless done so given time constraints.

5 Conclusions

This architecture uses components stored in game objects, which are demoted to simple containers, as well as singleton-like objects. A uniform grid is used as a spatial partitioning system as well as a "state saver", allowing for more efficient collision detection and decoupled wall visualization. An alternative option would have been to store the components in arrays directly in the `game` class, an array for each type of component. Then during the game, all the components belonging the same array could have been updated, effectively implementing *locality of reference*, as described in [3] Chap. 17. This could have been the reason why the Game class is a game object, as it would allow for directly having arrays of components. Much of this architecture can be improved on, such as making the Background Changer and Audio separate objects to divide responsibilities better. Certain implementations of features can also be improved, namely the player timer for snappier player input and the hard-coded nature of audio within the engine, which currently has a pointer to each audio file.

References

[1] std::set. `http://en.cppreference.com/w/cpp/container/set`. Accessed: 2017-02.

[2] Tron (video game). `https://en.wikipedia.org/wiki/Tron_(video_game)`. Accessed: 2021-03.

[3] Nystrom, B. Game programming patterns. Online, 2014.