Distributed Systems – Final Project

Lukas Häfliger
ETH ID 11-916-376
haelukas@student.ethz.ch

Alexandra Maximova
ETH ID 09-913-534
amaximov@student.ethz.ch

Thomas Müller
ETH ID 11-946-936
muellltho@student.ethz.ch

Christian Vonrüti
ETH ID 11-930-914
cvonruet@student.ethz.ch

Alexander Viand
ETH ID 09-940-131
vianda@student.ethz.ch

Marko Živković
ETH ID 10-921-211
markoz@student.ethz.ch

ABSTRACT

We present a cross-platform game called Tronium that allows up to eight players to play together via local network, or alternatively allows single-player matches against AI opponents. Tronium is inspired by the "light cycle" scene from the 1982 film "Tron" and is implemented using the Unity® engine, which is a high-level framework for game development. The game supports Windows, Mac OS and Linux on x86/x86_64 and Android™ with potential for easy ports to others platforms thanks to the cross platform capabilities of the Unity engine.

1. INTRODUCTION

We were inspired to create this game after playing Armagetron Advanced[1], an open source game that is itself inspired by the light cycle scene from Tron. In Section 2 we will look at Armagetron Advanced and other games inspired by Tron, as well as similar games available on mobile platforms. In Section 3, we will explain the gameplay and game mechanics that we decided to implement with Tronium, and in Section 4 we will introduce Unity with a focus on its networking concepts. In Section 5 we will show how we use these concepts in our game. In Section 6 we show our implementation of collision detection and prediction, ensuring reasonable behaviour in case of network delays that are 'long' in comparison to the high in-game speeds. In Section 7 we introduce the AI that we use in the single-player mode as well as our own extension to the game, randomly roaming obstacles. In the last section, we will give our conclusions on our game and working both with Unity as well as with a larger group.

2. EXISTING GAMES

Armagetron Advanced[1] is one of two open source projects that try to recreate the light cycle scene, the other being GLTron[2] by Andreas Umbach. GLTron is a faithful reproduction both visually as well in terms of game mechanics of the original light cycle scene from the 1982 film Tron. Armagetron Advanced is less faithful to the film and offers a wide variety of gamemodes and heavy customization, with parameters like base speed, arena size, number of players, etc. We were aiming more at recreating the enjoyment of playing Armagetron Advanced rather than being faithful to the original scene. Since the scope of this project was somewhat limited, we do not offer the extensive options and choice of game mode that Armagetron Advanced does, but rather focused on implementing the 'core' game mode with fixed parameters. Visually, we were inspired by the aesthetics of the 2010 film "Tron Legacy" rather than by the original film or the existing games.

Armagetron Advanced offers local (split-screen), local network and internet multiplayer. It is available for Windows, Linux and Mac OS but does not have an Android (or other mobile platform) port. There exists an Android game called Androgetron[3] developed by a member of the Armagetron Advanced community, however this is a much simplified clone without networked multiplayer. On the official Google Play™ Store there are a few more games[4][5][6] that are essentially identical to Androgetron and seem to be all based on an android port of GLTron. They also do not offer any kind of non-split-screen multiplayer.

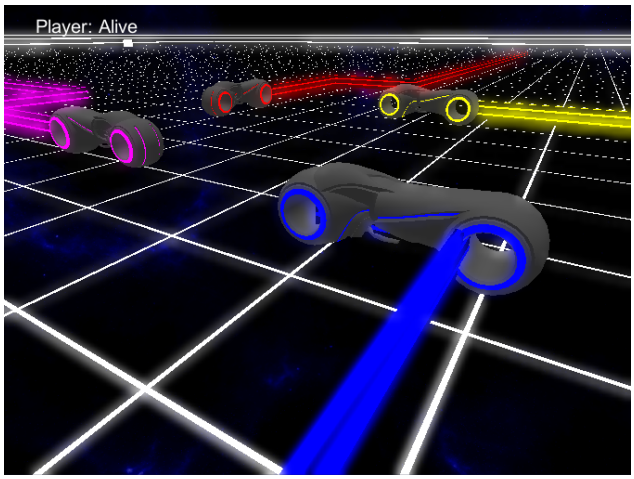

Figure 1: A screenshot from Tronium

Split screen is a viable option on a full size laptop or desktop monitor, however it feels very cramped even on larger tablets and is essentially unusable on smartphones. Our goal was therefore to implement a true (networked) multiplayer experience on mobile devices. Instead of trying to adapt the large and complex code base of Armagetron Advanced (or GLTron), we decided to implement a version of the game with smaller scope from scratch. We used Unity, a high level game engine (which will be discussed in Section 4) to implement the following game mechanics:

3. TRONIUM MECHANICS

We simplified the complex settings and mechanics from Armagetron Advanced to a single game mode, with fixed speed, arena size, etc. We did, however, also extend the game to new concepts not present in any of the other versions. Specifically, we introduced collectible "powerups" and moving obstacles that roam the arena.

The game takes place in a (square) arena, in which the players drive with their "light cycles" (from now on simply bike). The bike moves constantly and can only be turned 90° left or right. Each bike leaves a solid trail (wall), and collisions with either the sides of the arena, your own wall or another player's wall will result in your death. The aim is to avoid collisions and to be the last player alive. Being close to another wall increases your speed, allowing you to e.g. overtake and 'box in' another player. There are also two different types of powerups that spawn randomly throughout the arena. They grant a temporary speed bonus

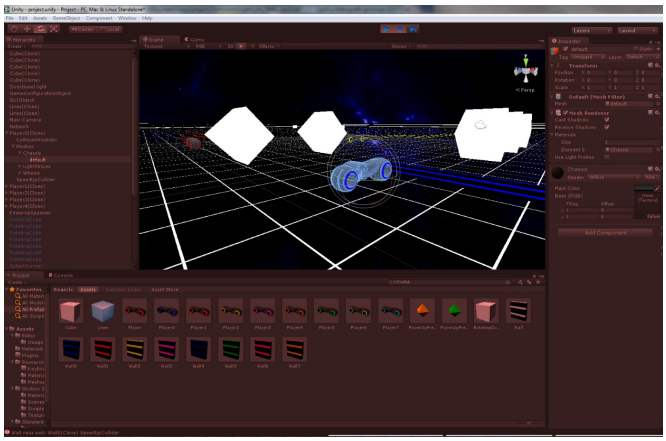

Figure 2: The Unity editor

or temporary invincibility (allowing the player to go through walls). There are also additional obstacles in form of large cubes that slowly 'roll' around the arena. Collisions with these obstacles also result in the player's death.

For single-player games, there are AI opponents which will take the role of other players (i.e. drive bikes as well). The AI and the roaming obstacles will be described in more detail in Section 7.

4. UNITY

Unity®, which is developed by Unity Technologies, is a high level game development framework or engine. Our project specifically uses the commercial version of Unity called Unity®Pro. Unity offers an editor (Figure 2) that allows manipulation of 3D environments, game assets and live debugging. Unity uses Mono (an open source .NET-compatible framework) and allows development in JavaScript, Boo or C#. We chose to use C# because of its similarities to Java and Unity's integration with the extremely powerful Visual Studio IDE. Unity supports a large number of platforms, including x86/x86_64 (Windows, Mac OS and Linux), Android, iOS and Windows Phone 8, with a generally very small required effort for porting between platforms. The support for native x86 versions greatly reduced the time needed for testing and debugging during development.

Unity's core concept is that of a `scene` which is an abstraction of a 3D space that contains `GameObjects`. `GameObjects` are containers that contain `Components` (which can also be `GameObjects`, allowing nesting). All `GameObjects` have a basic `transform Component` which contains information about position, rotation and velocity of the `GameObject`. There are many other types of `Components`, including `Lights`, `Cameras` and `Materials` as well as scripts.

Unity offers two very different methods for networking. A high-level "synchronization" feature as well as comparatively 'low level' Remote Procedure Calls (RPC). Both require a `GameObject` to have a `NetworkView` component which makes them visible to the network in Unity. Synchronization allows for high-frequency, low-cost updates of `GameObjects`. It can however only be used to synchronize certain properties of the `GameObject`, e.g. the `transform` (i.e. position,rotation,velocity) `component`. RPC on the other hand, is intended to be used for less frequent and less time-sensitive communication and is very similar to RPC implementations in other frameworks.

5. TRONIUM NETWORKING

Tronium combines (as do most Unity applications) both methods, using Synchronization for player and wall positions (or more correctly player and wall `transforms`), as well as RPC for communicating gamestate changes.

In terms of networked multiplayer, most games can be classified into either an authoritative or non-authoritative server style.

With a non-authoritative server, clients will do their own calculations locally and based on those will report e.g. their player's new position and events like the player's death to the server (and potentially other players). This is in contrast to an authoritative server, where the client is mostly reduced to relaying the user's input to the server which in turn does the necessary calculations and sends back information about e.g. the player's position and liveness back to the clients.

Authoritative servers have the advantage of preventing many common forms of cheating that involve running modified client code, however they also require a more powerful machine as server and result in additional latency.

Since our game is quite fast paced and supposed to run on relatively low-power mobile devices and will most likely not be subject to complicated cheating attempts, we feel that non-authoritative server was the best fit for our application.

Updates regarding the movement of the player and the creation of new walls in a player's trail are handled by the player instance and are communicated using Unity's state synchronization system. The system offers two options for data transfer. One is a unity-specific reliable data protocol that uses delta compression (i.e. sending only the information that has changed) and ensures in-order delivery. The other is unreliable transmission via UDP. Since in a racing game low latency is more important than in order-delivery, we use the UDP transmission mode. All clients will locally extrapolate (based on last known speed) the position of other players' bikes. In Section 6 we will explain how we deal with network delay when calculating collisions and player deaths to ensure that players will never be 'retroactively' killed because of information that arrives late.

Spawning of new `GameObjects` (e.g. walls) is done via Unity's `Network.Instantiate` method, which automatically creates the same object on all clients.

Less frequent updates like e.g. player deaths and game start/end are sent via RPC.

Since Unity does not have a high-level concept representing the current application instance in its entirety, every script has to be attached to a `GameObject` and each instance of the game will execute the same code for a given

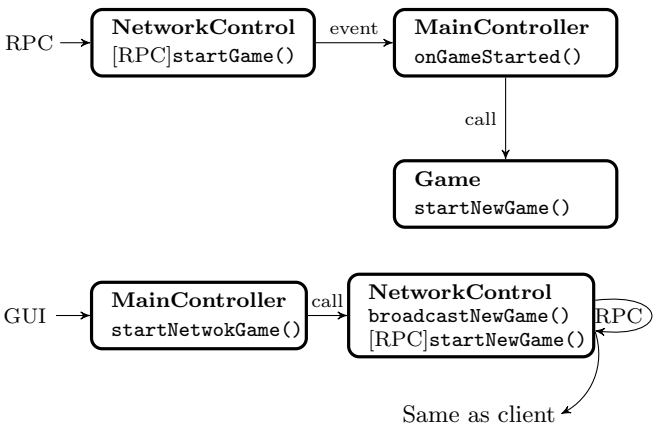

Figure 3: Control flow diagrams for client (top) and server (bottom).

GameObject. Since our server instance will also be acting as a player/client instance, this model works well with our application design.

Checks like GetComponent<NetworkView>().isMine() or Network.isServer() allow to test whether the current GameObject (or more specifically its NetworkView) is owned by the current instance and whether we are acting as the server, respectively.

The state transitions work as follows: There is an empty (i.e. code-only) GameObject that contains the MainController which is responsible for updating the GUI and (locally) communicating state changes between the Game and the NetworkController. The NetworkController handles all RPCs and causes events to be triggered (via C# delegates) in the MainController, which will then in turn call methods of the NetworkController or GameController. A received RPC will result in an event being triggered in MainController. The MainController calls (non-RPC) functions of the NetworkController which will then in turn send out RPC Calls to the network. In our implementation RPC's will also be sent to the sender, ensuring that all clients make the same state transitions. See Figure 3 for examples of (simplified) state transitions involved with starting a new game via network.

6. COLLISION DETECTION

Collisions in Unity can be implemented either with 'true' collisions or with Triggers. With Triggers, the objects do not actually collide, i.e. no forces are applied but instead the two methods OnTriggerEntry() and OnTriggerExit() are called when an object moves into or exits a Trigger. Collisions will be triggered during the physics update phase of each frame calculation and the effects of these method calls can be used in the Update() method that is triggered later (but still before the actual frame is drawn).

In Tronium, collision detection is handled by the owner of the bike, i.e. the game instance of the player controlling the bike. This means that players will only die to obstacles that are visible to them in their current local state and packets arriving late will not retroactively kill the player. The bike has three different types of Colliders (Triggers), shown in Figure 4. Each Collider is a child GameObject of the bike and has a script attached to it that listens (via OnTriggerEntry/Exit) for collisions. If a collision is detected, an appropriate event in the parent GameObject is triggered (via delegates).

The BodyCollider is the main collision mesh for the bike, and ensures that players cannot drive through each other's bikes. Using a simple shape like a box greatly increases the performance of the collision detections.

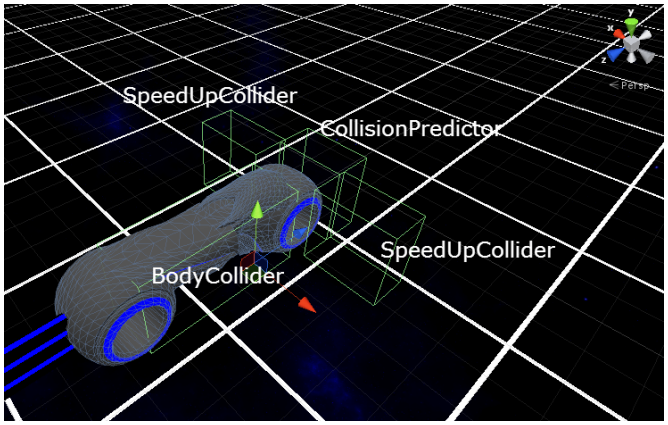

Figure 4: The player bike, with the colliders visible in yellow

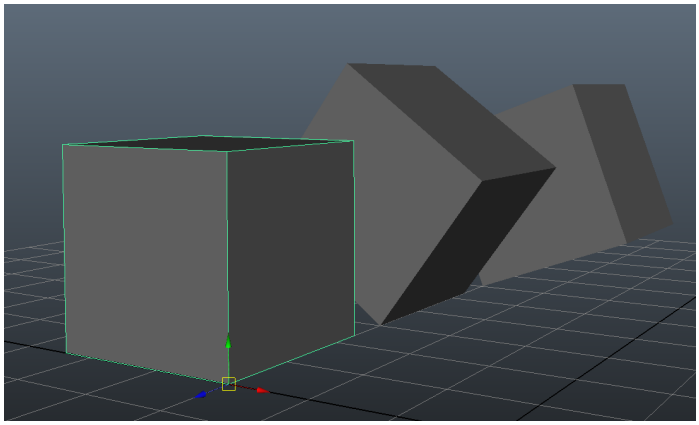

Figure 5: Demonstration of cube rotation

The SpeedUpColliders are used to detect when a player is close to a wall (which will increase his speed).
Finally, the CollisionPredictor is used to detect if the player will move into (or 'through') a wall in the next frame update. If that is the case, the player will be killed and the movement cancelled, unless the player has an active 'invulnerability' powerup in which case only the arena walls will have any effect on the player.

Placing the CollisionPredictor in front of the bike is necessary to avoid players 'warping' through walls - this could happen since collisions (and position updates) are calculated at discrete time points. With sufficient speed, a player might be before an obstacle at one point but already beyond the obstacle in the next. The CollisionPredictor is therefore dynamically resized to match the size of the next 'step', i.e. it covers the ground that the bike would move through during the current time step if time was truly continuous.

```
1  protected void AdjustSpeed () {
2      if (_numberOfWallsNear > 0) {
3        _speed = Mathf.MoveTowards (_speed, MaxSpeed,
             AccelerationRate * Time.deltaTime);
4      } else {
5        _speed = Mathf.MoveTowards (_speed, MinSpeed,
             DecelerationRate * Time.deltaTime);
6      }
7
8      if (CollisionPrediction != null)
9        CollisionPrediction.Length =
             _speed*transform.localScale.z;
10   }
```

Listing 1: Updating the CollisionPredictor

7. COMPUTER CONTROLLED OPPONENTS

In local single-player, there are computer controlled bikes that follow a simple pattern. Whenever a computer controlled bike predicts a wall collision (using the same code as above), it will do a "saving" turn (with equal probability for left and right turns). To ensure that they cannot survive forever, they are only allowed to do these "saving" turns every second frame. To avoid making the AI too predictable, there is also a chance that they will randomly (again, left right being equally likely) turn even if they do not detect any walls. This results in a very simple AI, however since the game is mostly about reflexes this unpredictable AI works well enough to make the single-player mode enjoyable.

Randomly moving cubes pose a more serious challenge. They move by rotating (relatively slowly) over one of their edges as depicted by Figure 5. After each rotation the cube will remain stationary for a set time, then choose a new

direction at random and continue to move in that direction. To achieve the movement as illustrated in the aforementioned image—assuming the pivot were in the center—the cube must be translated along two axes and rotated simultaneously over time. However, if we choose to rotate around the blue axis (which also represents time in the image), none of this is necessary. Fortunately, Unity provides a `RotateAround()` method to do just that. Our implementation can rotate only over one of the cubes edges. To allow movement in any direction, we simply rotate the cube around it's center before initiating the movement. This works because the cube's sides are uniform: a 90° rotation is, as far as the viewer is concerned, invisible (These rotations happen instantaneously between frames). This axis is specified by two vectors, an origin relative to the cube's center: {x: -.5, y: -.5, z: -.5 } and another vector to indicate the direction: {x: 0, y: 0, z: 1}. Unity cube primitives have an edge length of 1, which produces -0.5 used in the coordinates. However, `RotateAround()` expects the axis to be in world coordinates, for this we transform both the origin and the direction out of our local coordinate system again leveraging Unity's built-in methods. Another benefit of this method is, that the scaling factor imposed on the cube is irrelevant to the movement and rotation, because of local-to-world transformations that are used. The code for the rotation is unique in that it makes use of a feature not present in the rest of the code: It uses a coroutine, or `Iterator` It allows us to program the animation inside a loop by passing control back to Unity using `yield return null` after we've made the calculations for the next frame.

On the next frame, Unity resumes our `Iterator`, thereby restoring all the local variables to the state in which we left them. Otherwise, we would need to keep the state within fields of the script, also keeping track in what stage we currently are. Using `Iterators`, however, the compiler does the heavy lifting while the code remains legible. This is also exemplified by the code used to wait for as many frames as needed until the movement is to be continued. Due to time constraints, cubes are allowed to move through any geometry. As an improvement, one might consider to have cubes tear down pieces of walls they come in contact with.

8. CONCLUSION

Working in a larger group was an interesting change and while our intra-team communication was very good, there were considerable efficiency losses when e.g. bug fixes for one component affected code that was concurrently being refactored.

We feel that we accomplished our goal of creating an enjoyable networked multiplayer version of a 'Tron-like game' for mobile devices. Unity allowed us to very quickly create the visual aspects of the game and allowed us to focus on the distributed networking component of the game.

An obvious and straight forward next step would be allowing the player to customize the parameters of the game. Another interesting extension would be internet multiplayer, however additional work for NAT and higher latencies might be necessary.

9. REFERENCES

[1] Armagetron Advanced. `http://armagetronad.org/about.php`. Accessed on 20 Dec 2013.

[2] GLTron. `http://www.gltron.org/index.php`. Accessed on 20 Dec 2013.

[3] Androgetron. `http://forums3.armagetronad.net/viewtopic.php?f=4&t=23073`. Accessed on 20 Dec 2013.

[4] "3D Tron Light Cycle" on Google Play. `https://play.google.com/store/apps/details?id=mobi.pixi.glTron`. Accessed on 20 Dec 2013.

[5] "TRON 3D" on Google Play. `https://play.google.com/store/apps/details?id=com.msi.tron3d`. Accessed on 20 Dec 2013.

[6] (simplified) GLTron port on Google Play. `https://play.google.com/store/apps/details?id=com.glTron`. Accessed on 20 Dec 2013.

[7] Androgetron on Google Play. `https://play.google.com/store/apps/details?id=net.guru3.androgetron2`. Accessed on 20 Dec 2013.

[8] MSDN Reference: yield. `http://msdn.microsoft.com/en-us/library/dscyy5s0.aspx`. Accessed on 20 Dec 2013.