



## Interim Report - Fruit Smashers

Nemanja Bartolovic, Matej Hamas, Daniel Keyes, Delio Vicini

April 25, 2016

# 1 Current progress

Overall we think we are on track and should be able to achieve everything we initially envisioned. We are almost completely done with the functional minimum and the low target, only one or two power-up models are still missing, since they have not been added in the code yet. We are also making good progress with the desirable and high target, even though there are still a few things missing. The main things we need to do to finish the game are:

- Implement a visually appealing fruit stand demolition. This is very important, as it is the connection of our game to the theme.
- Fix the car physics: currently the cars flip too easily, which maybe negatively affects player experience. For example, one can quite easily get stuck.

Then what remains are smaller tasks from the schedule and continuous AI improvement. The detailed schedule is shown in Figure 1 on the following page. Some screenshots of the current state of the game are shown on the following pages. Some more development details and challenges are described in Section 2 of this report.

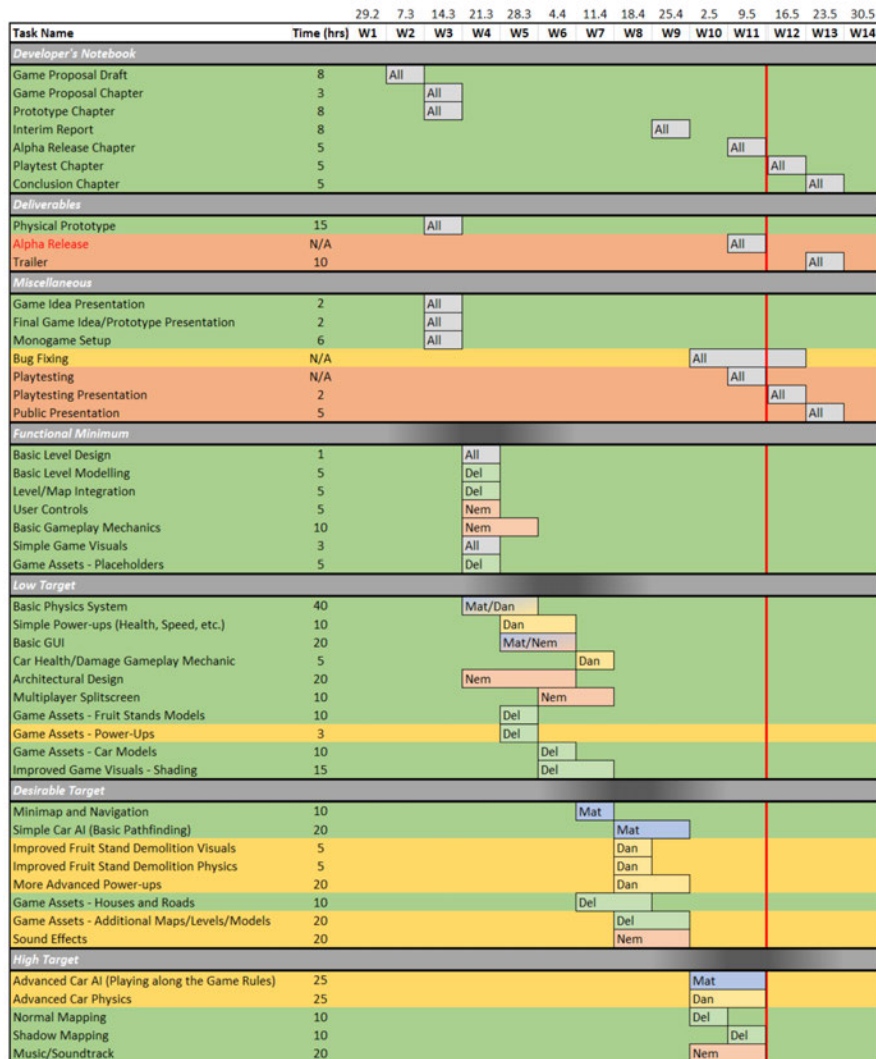


Figure 1: Screenshot showcasing the current state of our game. Green items are finished, yellow ones are currently in development and red ones are not yet started.

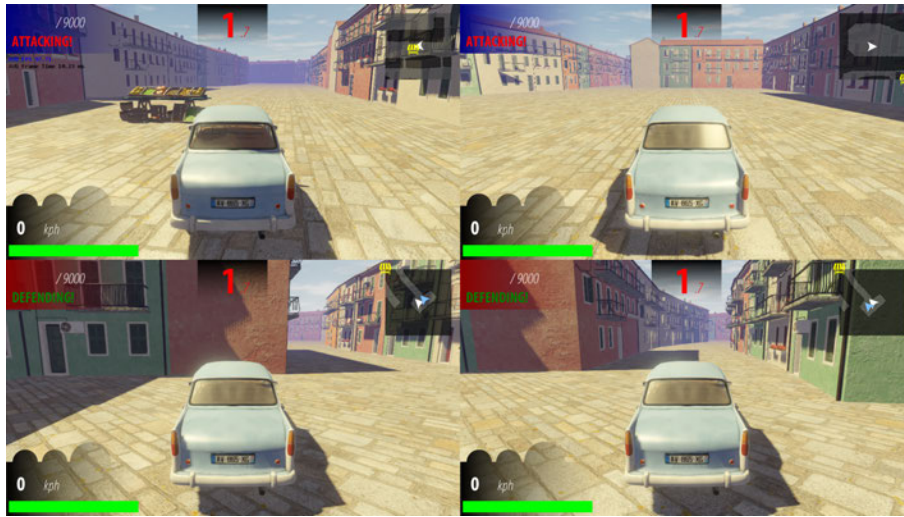


Figure 2: Screenshot showcasing the current state of our game.



Figure 3: A speed power-up in the game world.



Figure 4: The main menu. In the final version, the background will be replaced by a nice screenshot.

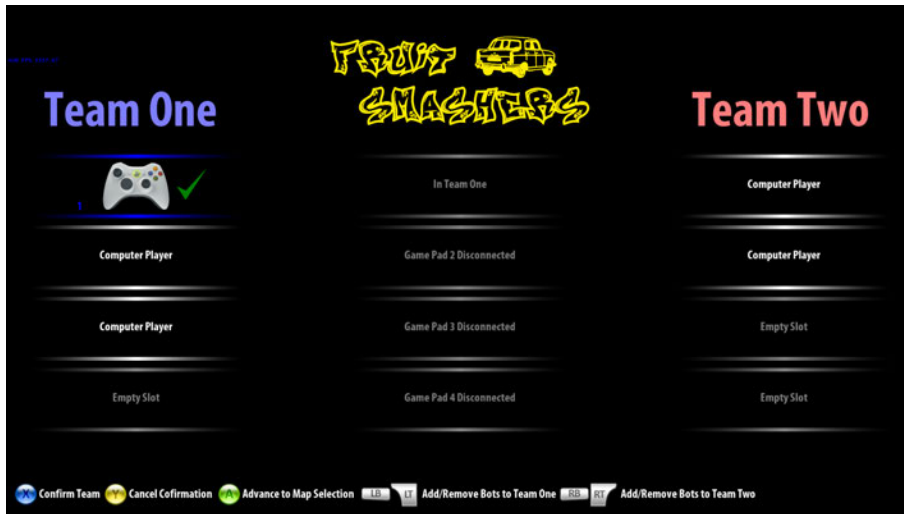


Figure 5: The team selection menu, which also allows adding AI players to both teams.

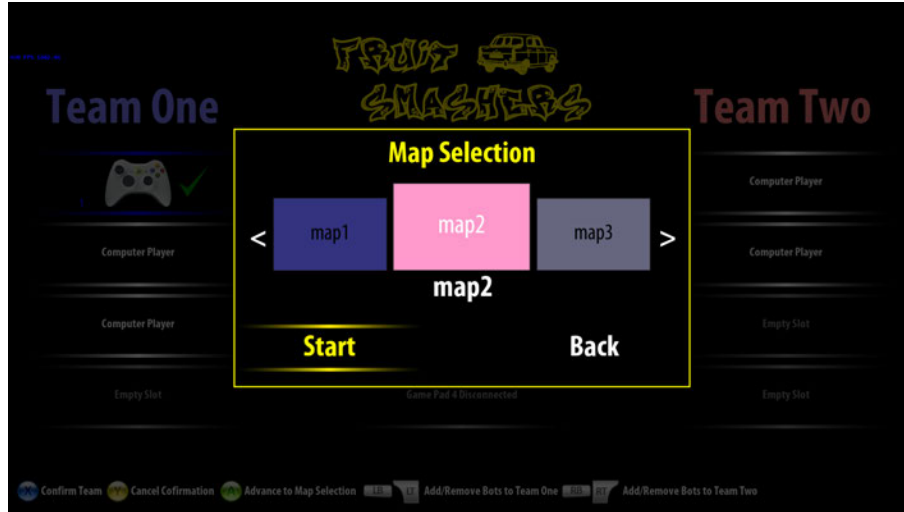


Figure 6: The map selection menu.

## 2 Development Details and Challenges

### 2.1 Rudimentary Application Modules

#### 2.1.1 Core Application Flow

The core flow of the application is architecturally designed after the activity stack found in Android architecture. The main idea is to have a stack of game states, where the top one is being updated and drawn, while being initialized, suspended and destroyed during its lifetime. When the state is added to the stack, it is allowed to initialize itself as well as grab content from the game, which is for simplicity, currently fully loaded into the monolithic Game object. When the stack is empty, it's time to close the application. Activity diagram is indicated below.

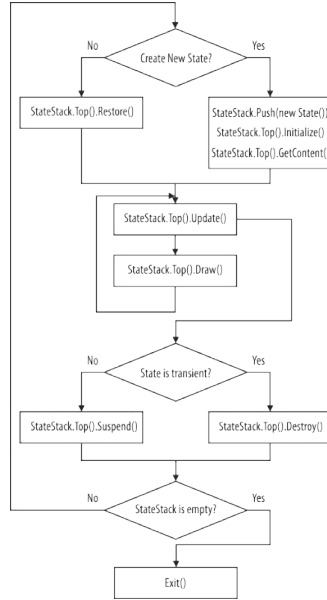


Figure 7: Application state flow

### 2.1.2 Entity Component System

The overall architecture of objects in the game loosely follows entity component principles, where different aspects of an entity are wrapped into separate components and then processed and managed by corresponding systems, e.g. Rendering or Sound components processed by RenderContext and AudioSystem, respectively.

Current system allows for fairly flexible additions of new entities, with different properties with regards to physics, visual appearance or sound.

### 2.1.3 Animations

The animation system is modeled to be very lightweight, allowing the system to be embedded into the objects itself where they can manage their own animations, or subscription-based where animations can be queued into the update calls that the current state receives.

In our game, we did not find any advanced animation requirements. As such, the current system has only PropertyAnimations which allow flexible single property (float, int, Color, Vector, etc.) updates with up to 15 different easing types (Linear, SlowIn, SlowOut, etc.). This part is fairly extendible with regards to properties that can be animated, as well as easing types, as long as basic arithmetic operations are available. Besides that, things such as duration, delay and number of loops can be easily specified.

In addition, the system allows for SequentialAnimations, which essentially serve

as key-frame animations that can chain up several animations and perform them in sequence, without worrying about correct delay setups.

#### **2.1.4 Audio**

The current audio module supports registration and playback of various sound effects, either in looped (like a car engine), or immediate one-time mode (GUI sound effects and similar).

Even though MonoGame provides useful interface for creating 3D sound effects, playing 3D-aware sounds is a fairly challenging aspect, due to split-screen multi-player gameplay. There's no clear consensus on what exactly to use in such situations, but one common practical solution seems to be to position the sound (by adjusting pan and volume) according to the closest audio listener (listeners are essentially all player cameras), which is the principle that we'll follow in our sound design.

The system also supports song playbacks for background music, with multiple playlists available and shuffle mode. A playlist is picked according to the current state. There are currently Menu and In-Game playlists.

#### **2.1.5 Gui**

The gui system that is currently in the game is modeled after WinForms / Java Component model, which allows for easy extension and addition of new elements. It is fairly easy to integrate with animations and extend with new widgets.

#### **2.1.6 Input**

There are essentially two ways that a game can process user input: either by polling or through events. Since event-driven design is favored in C#, but not exactly provided with MonoGame input interface, we've built a system on top of that one that provides event-based input and allows for minimal effort when adding new human interface devices to the system.

Overall architecture is divided into three layers:

1. The low-level system that queries MonoGame input states and transforms them into relevant events (e.g. ButtonPressed, ButtonDown, ButtonReleased and others).
2. The key binding layer that wires together low-level events with corresponding game actions, states and ranges. This layer is responsible for key-binding and can be loaded from a configuration file (we use xml).

3. The input mapping layer, which is responsible for calling delegates that were registered for every game action. Note that this layer has no notion of underlying input device or actual button pressed, which allows for full input abstraction.

Range-based input is also implemented for fine-grain control over certain actions that support it (e.g. steering amount with thumbsticks).

The design of this system is based on the concepts described in <http://www.gamedev.net/blog/355/entry-2250186-designing-a-robust-input-handling-system-for-games/>.

## 2.2 Graphical aspects

### 2.2.1 Progress

We already modelled the car, the fruitstand, several house variants and some of the power-ups. We also implemented a system to author levels in Maya (using Maya references), from where we can export a level description as a text file to then load the appropriate models at runtime.

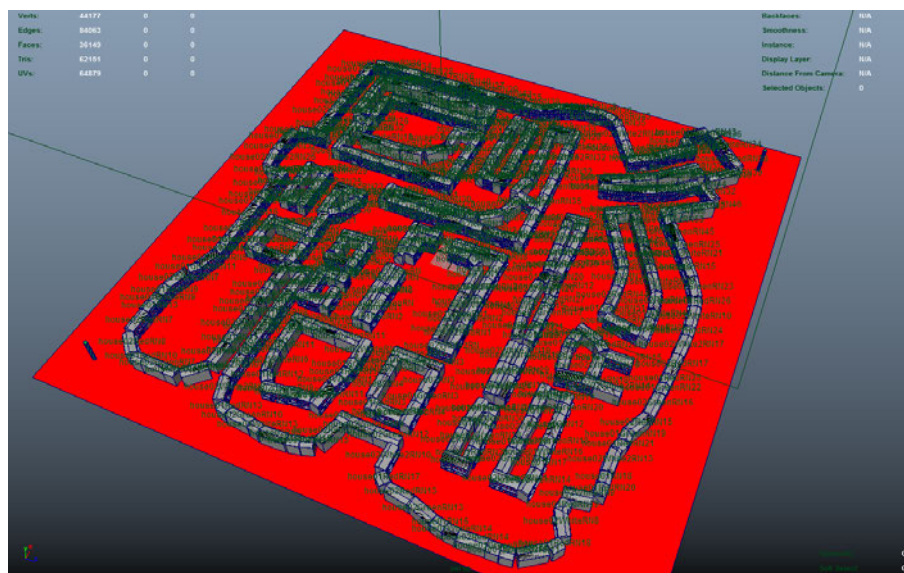


Figure 8: Screenshot of the current level in Maya. We had to use low poly proxy objects for our houses, since Maya’s performance with references is quite bad.

The current rendering pipeline supports phong shading, including normal and specular mapping as well as simple exponential fog. Also alpha blending and cutout opacity are supported. The alpha blending is used for the car windows and the cutout opacity for the balconies. We also implemented cascading shadow mapping, currently using 2 shadow cascades of  $1024^2$  pixels each and percentage

closer filtering. Furthermore we added high dynamic range rendering, bloom and filmic tone mapping.



Figure 9: Screenshot showcasing the graphical fidelity of our game.

### 2.2.2 Challenges

The main challenge we faced was performance. Our current level has over 400 individual houses, adding up to roughly 600K triangles. Therefore just brute force rendering all objects does simply not work. We thus first added view frustum culling to avoid drawing objects which are not visible by the camera. This already helped quite a bit, but we were still limited by the number of draw calls we can issue. Note, that since our game runs in 4 player split screen, the number of objects drawn is 4 times as large as in an equivalent single player (or single screen) game. Also shadow mapping adds on top of that, especially in the case of using more than one shadow cascade per player. Since view frustum culling alone was not giving sufficiently good performance, we then also added hardware instancing, after considering several other options. This gave again a significant performance increase and we hope that no major further optimizations are necessary.

## 2.3 AI vehicles

### 2.3.1 Progress

The AI vehicles are capable of playing the game, both as attackers and defenders. When the vehicle attacks, it drives to the fruitstand as quickly as possible and tries to reach it at a high velocity. For now, when it's playing the defending role, it stops nearby the fruitstand, blocking the street.

Given the top down view of the map, we have created several Matlab scripts that allow to conveniently specify points and edges of interest using a mouse. This enables us to build an abstract graph that is used to

- guide AI driving.
- define the spawn locations of fruit stands, power ups and the initial team spawn locations.

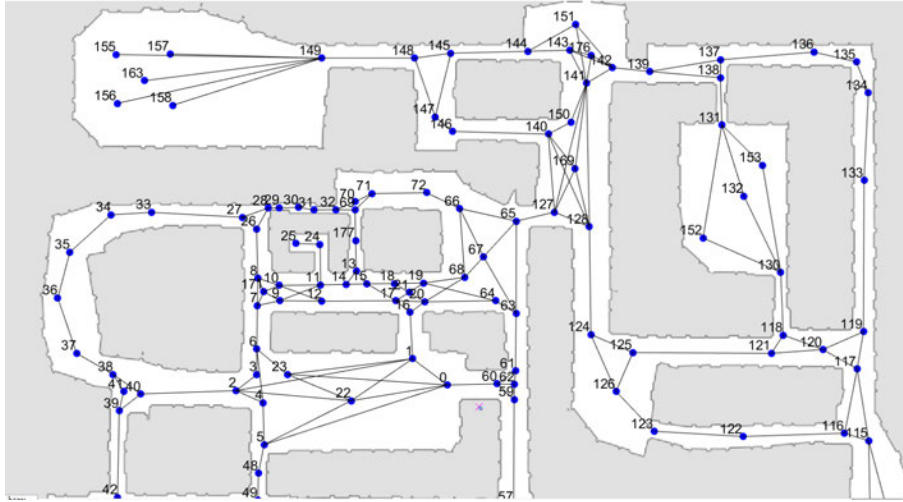


Figure 10: Screenshot of the graph created on the top of the current map.



Figure 11: Screenshot of the graph visualized in the game itself. We've added this feature for development purposes.

Given the target vertex, the AI vehicle computes the shortest path from its position to that vertex using the Dijkstra algorithm. It then drives along this path, trying to drive as fast as possible while avoiding collisions.

We have implemented heuristics to be able to drive fast and slow down just before the turns. Also, when a crash is detected, the car reverses a bit and then tries to get onto the original path. If this is not possible, we respawn the car on the road nearby the location where it crashed.

Some heuristics, such as the distance needed to brake to the zero speed starting at a given velocity, are based on experiments. Others, such as the crash detection, make use of the ray tracing capabilities of the BEPU library.

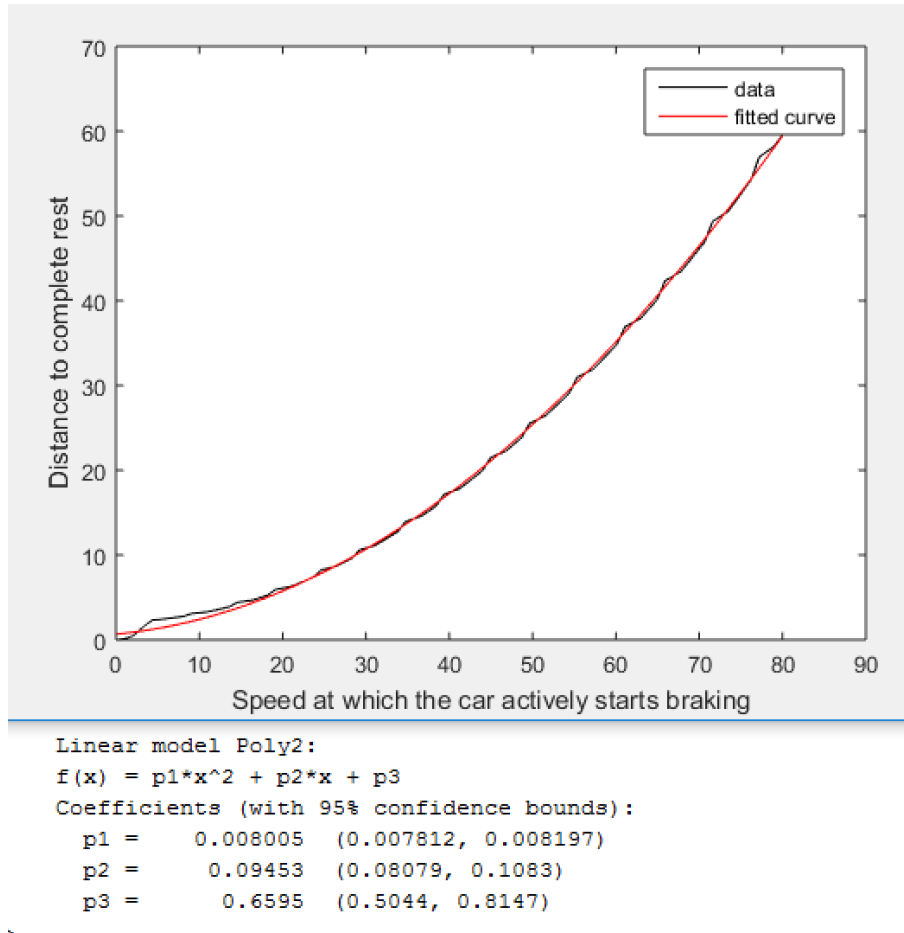


Figure 12: Given the starting speed (x-axis), the y-axis shows the distance needed to reach complete rest when the vehicle actively brakes. The data from the experiment are shown in black and the fitted quadratic curve in red. The fitted parabola is used in the code to estimate the optimal current speed. The AI vehicle either accelerates or brakes to reach this speed.

### 2.3.2 Challenges

The main challenge is to minimize the number of crashes while driving as fast as possible. This means accelerating when the street is straight and braking in front of the crossroads. Since the streets have different widths and crossroads different angles, this is not trivial. So far, we have put considerable effort into this part and the performance is quite satisfiable. If it turns out that the AI cars still crash too much, we will attempt to further optimize their driving behavior.

### **2.3.3 Future Development**

Next steps will be to modify the defending mode where the AI vehicle will not just stop nearby the fruitstand, but will move back and forth to make it more difficult for attackers. After that, we will concentrate on making AI vehicles trying to crash the opponent.

## **2.4 Physics**

### **2.4.1 Progress**

The core simulation of our game is handled using the BEPU physics library. This gives us a number of useful components out-of-the-box, including vehicle prefabs, shape primitives, containers for meshes, and collision detection and solving for those previous components. Vehicles, scenery, and some debris in the game are linked to BEPU entities to provide realistic physics. Fruitstands do not go through the full pipeline, but rather use BEPU to detect collisions for triggering game objectives.

### **2.4.2 Challenges**

One challenge with BEPU is achieving a stable simulation. With very large numbers of objects (like stacked fruit meshes), the simulation becomes both unrealistic and sluggish. Currently, we address this by showing simple animations instead of properly simulating complex fruit-based physics. In the future though, we plan to tweak the collision rules (to disable some interactions) and collision shapes (to simplify other interactions) to achieve more realistic fruitstand destruction.