02825 Introduction to Computer Game Prototyping E07
Final Project Report

Jakub Cupisz s061361
Robert Cupisz s070924

December 17, 2007

1 Introduction

Author: Robert

The aim of the project is to gain confidence in using the Squeak development environment and software prototyping design approach, in the process of developing a simple game.

This paper describes the game concept, it's analysis and design, our programming approach and resulting implementation. Report is concluded with critical assessment of Squeak as a prototyping tool.

2 Game concept

The game is yet another clone of the so-called Tron or Light Cycle Race. The original idea came from the Tron 1982 Disney science fiction movie, in which the main character along with five others, are racing in futuristic bikes, called light cycles. Light cycles cannot be stopped, can make only 90° turns and leave walls behind them. In the movie sequence, the arena of the race or the grid, quickly becomes a labyrinth and some of the bikes are forced to run at high speed into previously left walls, which results in spectacular explosions.

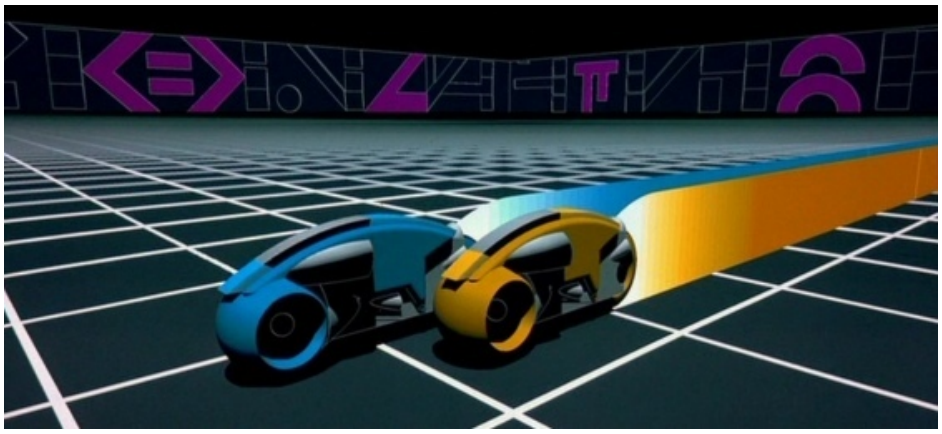

Figure 1: The Tron movie (1982) light cycle race scene.

2.1 Basic rules

Our version of the Tron game is a two-player arcade type game with two bicycles racing on a two-dimensional grid or plane. The area is bounded by four walls. Each bicycle can go only along one of the two directions (north-south or west-east) and leaves a permanent wall behind itself. Bicycles crash

by running into any of the walls present on the game board: the bounding walls, a wall built by the opponent's bike or even it's own.

Bicycles start facing each other and already running at high speed. A bicycle cannot stop without crashing into a wall — it can only go faster by exploiting a bonus.

2.2 Goals

The goal of the game is to make the opponent's bike crash before our bike does. It can be achieved by building a wall around opponent's track and blocking him in a clever way. If one bike hits the other, then the round ends as a tie. There are up to five rounds and wins that player, which achieves three victories first.

2.3 Bonuses

Each player has three turbo bonuses at his disposition for each round, which provide a great increase in the bike's speed, but only for a short period of time. Another way to speed up the bike is to run along (and very close to) any of the walls present. Those bonuses create more possibilities for inventing new strategies.

2.4 Controls

Bikes can turn right or left by 90° or fire the turbo. Both players have a group of three keys on the keyboard at their disposition.

2.5 Graphics

All of the game environment is displayed as an isometric view of a 3D game board. All the walls are drawn in a semi-transparent way, so that a bicycle runing behind is clearly visible. It also aids the player in estimating how narrow is the corridor between two parallel walls, if he's planning to fit in it. Wall gets even less opacity if the bike was riding faster while leaving it behind.

Bikes, explosions, etc. have their representation in the isometric view. A bike throws a fountain of sparkles into the air while riding along a wall.

2.6 Single player

Optionally an AI module will be developed, so that single player will be able to play against the computer.

3 Analysis and design

Author: Jakub

This project was created in the process of design approach known as prototyping. Thus the analysis and design phase of the development process is somewhat degenerated. It identifies the indispensable elements of the game, some of the early-stage ideas and ways to implement them.

Main emphasis is put however on the iterative process of developing as quickly as possible a working prototype of the game, testing it by ourselves, testing it by other other players, identifying improvements and making them work with the game to be able to test it again.

3.1 View of the game board

As stated in the game concept section, players will be viewing all of the game environment in an isometric view of the whole grid. While this is a much more interesting way of viewing the action than the birds-eye view, when seeing the walls as plain lines, it causes some issues, that have to be solved.

The isometric view has also some advantage over the typical first-person (or first-bike) perspective seen in many other Tron games. While not so visually thrilling, it lets the players easily see the situation on the entire grid, and so the game a more strategic than action taste.

3.2 Drawing in the isometric view

All the coordinates will be stored, as if the game-board was viewed from the top. Thanks to that, objects will be described as rectangles defined by their position and extent, having only vertical and horizontal edges. Each object class can then be simply a subclass of Morph, which calls this attribute *bounds*.

To achieve the impression of a 3D game environment, objects' drawOn methods will be modified. New method will take x@y coordinates of each corner of the bounds attribute, and compute new values for the isometric view in a form: (x-y)@((x+y)/2).

After such conversion the grid itself appears as it would be tilted to the right 45° and then scaled down vertically by 0.5, which gives the impression of observing the grid from some height above one of it's corners.

3.3 The walls

The walls are the most important element of the game. They have to be drawn in such a way, that user gets visual hints about their localisation and orientation in 3D world. During the action players will have to be able to quickly assess, if their bike will hit the corner it's approaching to, or pass just by it. To achieve that, some walls will have to occlude others, standing close, but behind them. Also when the bike turns, newly created wall will have to be displayed either in front of or behind the previous wall, to show that the bike is going to or away from the viewer.

Each wall is a morph, and it's new coordinates are computed like described in the previous section. Walls have always the same thickness and are either horizontal or vertical. New drawOn method for drawing walls gets isometric coordinates for origin and end of the wall and shifts them up, by the desired height of the wall. This four points are the basis for drawing a polygon representing the wall.

The proper occlusion of the walls was suppose to be achieved by sorting all the walls present in the game and then drawing them in that special order — a form of the painter's algorithm. During the implementation we run, however, into several problems resulting from that approach. The whole process is described in the section 4.3 The wall occlusion problem.

3.4 Race organisation

Probably the whole competition between two players will be organised in rounds of few races. Player wins a round, if he wins the majority of races in that round. Game board displays information about the number of victories of each player. This should be done in a form corresponding to the overall look of the game, i.e. using some objects viewed in isometric. Precise organisation of the game will be worked out during the tests, with the aim of improving playability.

3.5 Additional effects

The game has to incorporate some additional 3D effects, which are supposed to add an eye-candy to the action. Those listed in the game concept section are fountains of sparkles thrown into the air, while bike rides along a wall and, obviously enough, explosions.

Sparkles can be implemented with a subclass of Morph with it's own four subclasses for four directions a bike can go: up, down, left and right. Each of those subclasses will have it's own drawOn method, drawing some

short, random lines being thrown away from the bike position to the opposite direction, than the bike is heading to. Sparkles can be drawn as a front-most morph in order to assure proper occlusion. Sparkles can also implement the step method in order to be animated.

Explosions appear in the game when bike hits a wall or bikes hit each other. They can be drawn as it was seen in the movie: as a flash running away radially from the site of the crash, in the plane the bike crashed into; or, in the case of two bikes crashing one into another, in the plane perpendicular to the direction the bike was going. We decided, however, to draw the explosion as a quite abstract shape, as it fitted to the overall game style.

3.6 Turbos

Turbos add more strategy and action to the game. If the player has only a few turbos at his disposal, he will try to use them wisely in the key moments of the race, in order to gain advantage over the opponent.

When engaged, turbo will significantly increase bike speed for a period of time no longer than some fixed amount. Player will be able to break the boost in order to get easier control over the bike again, but the remaining time in that bonus unit will be wasted.

When the bike speeds up, the part of the wall, that he was generating while speeding, gets different color or opacity. This way player gets more visual feedback, that he has successfully engaged turbo and the other player may notice, that he is in trouble.

Bike also speeds up when goes very close to another wall. This turbo does not consume number of bonuses available for the player, but cannot be switched off in other way, than turning off the wall (if it's possible). Approaching a wall can be dangerous, as it is very easy to hit, when not being careful. This will give more skilled drivers some additional advantage. Besides changing wall colour, bike throws some sparkles into the air, while going along a wall, as another hint, that it successfully approached a wall and is gaining speed.

During the tests we will have to quite carefully choose the number of turbo bonuses available for each player, maximal duration and amount of speed increase. Choosing those figures incorrectly may cause, that players will not be paying too much attention on using them and they will not have significant meaning during the game.

4 Implementation

Author: Robert

This section describes our development approach, then roughly the class structure we came up with during the implementation, and finally we discuss some of the problems we have had with the application and the platform.

4.1 The process

We were using the pair programming technique during the majority of the time spent on the project. This worked out quite well, provided that we had to rapidly create pieces of really simple code. When one person was writing the actual code, the other could look at other class browser windows and plan next changes to the code, or look for any bugs in the newly created code.

It is a hard task to correctly assess in advance all the features a game would need to be playable. That was the reason why we have chosen the Prototyping design approach to guide development of the game. As quickly, as it was possible, we were creating a working version of the game, and on this basis we could rethink our initial design ideas. The new ideas were the basis of creating next prototype, and so on.

In the Analysis and Design section we have not described specifically any class structure, we were planning to create. This was the result of the Prototyping design method and the fact, that we wanted to use the same work-flow, we have previously tested while following the *Development Example for Squeak* [2].

To create working prototypes of the game (or, initially, some elements of it), we were creating only the most essential objects, that would do the work, and then we would try using them in the environment. When we already knew, how to use the (partial) functionality of a newly created object, we were correcting the previous class structure, so it could then acquire the new class.

This iterative process was driven by constant manual testing of the game being developed. We have not used the testing capabilities of Squeak, because the major part of the code is either the graphics, either it is simply hard to test using a short piece of code. Thankfully, Squeak dynamically reflects nearly all the changes a programmer makes in the code, without the need of relaunching the application, so the process was truly rapid.

4.2 The class structure summary

Author: Jakub

Every class in the game code is a Morph subclass, because every object uses at least some of morphic functionality: either it is drawing, attaching as a submorph or parent, stepping or event handling.

The entry point to the application is a Tron class instance. It can be created by executing in workspace the following code snippet:

```
t := Tron new. t openInWorld.
```

`Tron`'s responsibility is to manage subsequent rounds and races, display information like welcoming screen, controls, game over box and points obtained during the current round.

To start a race, `Tron` creates a new `TronRace` intance, which is automatically filled with all the submorphs, that belong in the race: `TronGrid` with it's outer `TronWalls`, `Bikes` with their `TronWalls` initialised, and `Turbo` bonuses for both players. `TronRace` manages collisions, receives event messages from `Tron` and sends them as commands to `Bikes`.

While moving, `Bikes` modify the current wall by extending it in the direction, they are going. After turning, a new wall is created and attached as a submorph of `TronRace`, to be properly drawn. Horizontal walls obtain a slightly different colour than vertical ones, to achieve a simple shading effect. A different colour is applied also to the new wall, that is being created, when a bike engages turbo — here it gives additional feedback to the user, than the bike is speeding really fast.

The game was initially drawn in a flat view with all the morphs having only vertical and horizontal edges, as described in the previous section. To obtain the impression of an isometric view, each morph has it's `drawOn` method overridden. New method sends a message to the `TronRace`, asking about conversion of old coordinates into isometric coordinates. After receiving an answer, morph can draw itself in a customised, 3D-like look by taking into account it's new coordinates.

4.3 The wall occlusion problem

Author: Robert

As the walls are the key element in the game, we wanted to make them especially fine-tuned. The original idea was that bikes should be able to run pretty close side-by-side, and the walls should be quite high compared to the distance — just like in the movie.

This would allow us to introduce three interesting effects into the game:

Maze-like look When walls are being drawn very close to each other and a bike enters a tight corridor with many corners, the grid starts to look like an interesting, dynamically created, labyrinth. We have seen that

effect in the early stages of development, of course with several wall occlusion artifacts.

Translucent walls When a bike runs behind a wall, it would be good to make the wall semi-transparent, in order to see, what is happening there. It also improves the overall looks of the game.

Turbo while close to other wall If going close to other walls is allowed, it opens the possibility to introduce this additional bonus, which was already discussed.

All that said, we were encouraged to find a good solution for the wall occlusion problem, which is really visible, when walls are high and close to each other.

4.3.1 The Painter's algorithm

Our first try was the painter's algorithm. We know, that walls never intersect and there is no possibility for a cycle of occlusion, i.e. A partially occludes B, B→C and C→A. This implicates, that for any given setup of walls on the grid, there is always at least one correct order of drawing the walls according to the painter's algorithm.

To achieve proper display with no visibility artifacts, we would need to find that ordering and make all the walls follow it, as they form a list of submorphs of the TronRace object.

The problem is that, for some pairs of primarily non-occluding walls, the order is not defined a priori, i.e. ordering the walls AB or BA would still cause them to display properly. Then a new wall can be drawn, which (for the fixed initial setup of the two walls A and B, of course) can possibly be in front of A and behind B, or in front of B and behind A. This means, that whatever initial ordering we choose (e.g. AB), the player can still draw a third wall, which makes that ordering invalid, e.g.: for newly introduced conditions C→A and B→C, the ordering AB has to be changed to BCA.

This implicates, that adding a new wall to the list of submorphs, is not a simple insertion to an ordered list problem, which in this implementation would take $O(n)$ time. Here adding a new wall can cause the whole ordering to become invalid. In that case list would have to be reordered from scratch. We believe, that this problem is NP-complete, as it looks like it could be reduced to finding a Hamilton path in a directed graph (which is proved, to be NP-complete), we have not tried the conversion, though.

That was the reason, why we could not find an efficient solution for a problem of size 20 to 80 walls. What's more, the re-ordering would have to

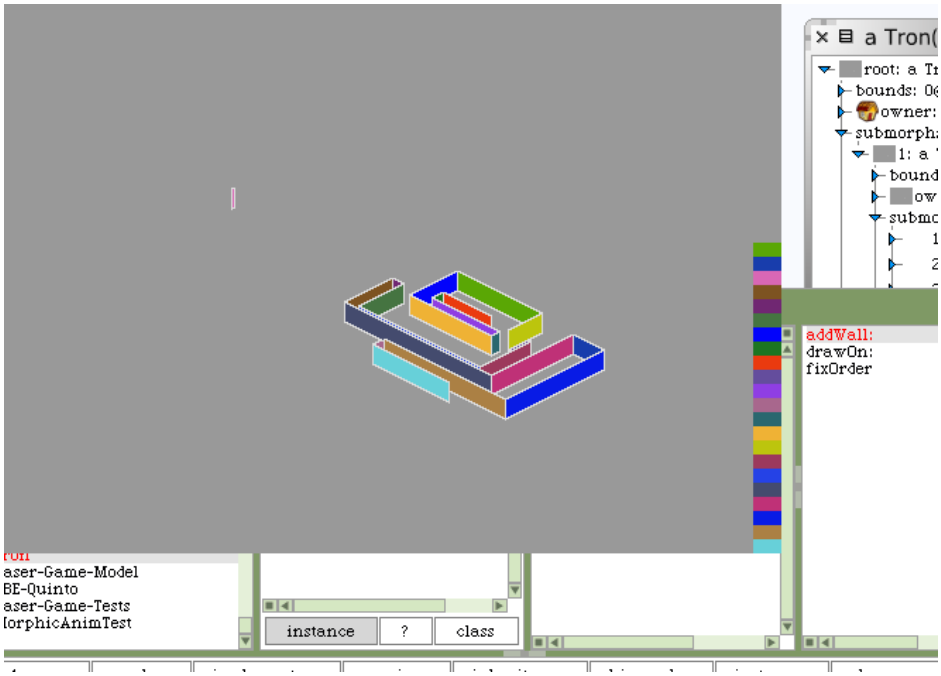

Figure 2: A test of the Painter's algorithm. For debugging purposes, each walls gets a random colour assigned. The resulting stacking generated by the current implementation of algorithm is shown on the right side of the game board.

be performed after each frame, as potentially situation on the grid changes with each frame.

4.3.2 The Z-buffer algorithm

This led us to developing a form of Z-buffer algorithm. The atomic element in our game is not a pixel like in the original version, but rather a whole vertical stripe of a wall, with width equal to one game unit. We were hoping, that this would be a sufficient performance difference, that would let the game run in squeak vm.

The z values for each part of the wall could be easily computed using an equation in a form $z = -x - y$. An object corresponding to a fragment of wall would then be inserted in time $O(n)$ in the proper place of the list of objects ready for display, according to it's z value.

We have implemented a test version of this algorithm (not using z-values yet), to test performance of squeak under that load. Wall pieces were not Morph subclasses, but simpler and lighter objects with as fast drawing methods, as it was possible. Screenshot captured during one of the tests is shown

on figure 3.

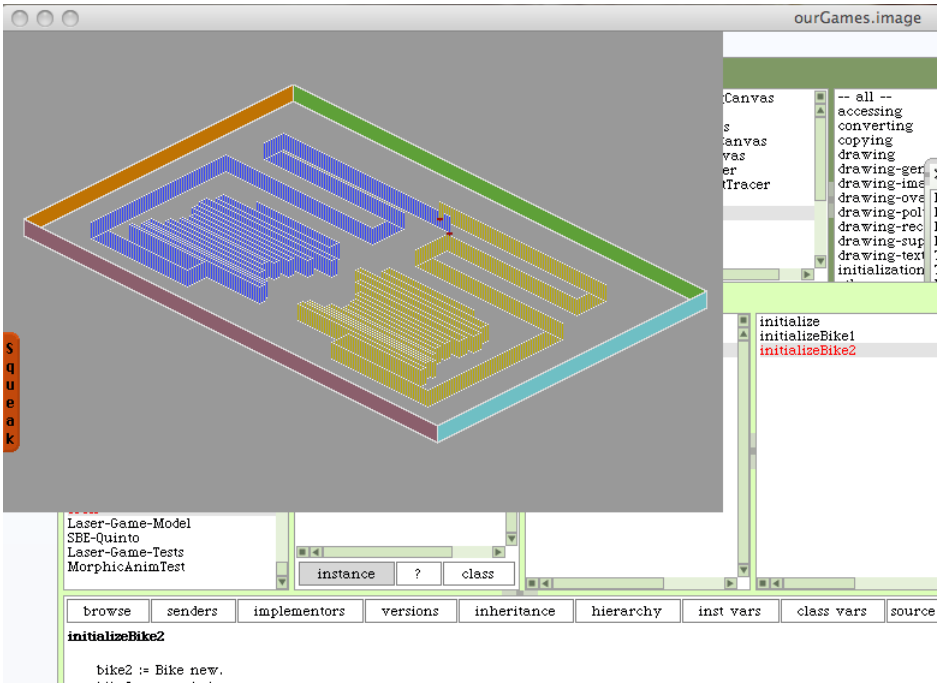

Figure 3: A z-buffer test. Each wall piece drawn as one vertical line (should be two in this scale). No calculation of z values yet, so occlusion artifacts can still be observed. Frame rate very low.

Unfortunately, platform was too slow for that solution, and after drawing a few wall, bikes were slowing down so drastically, that it was no longer possible to play.

Thus we had to retreat to a simpler solution, which would satisfy all of our requirements.

4.3.3 A simplified Painter's algorithm

The simplified solution we have finally chosen, was that walls would never be drawn so close to each other, to cause occlusion. The only exceptions to that were the cases of corners and outer grid walls.

If a bike is turning away from the viewer, a little part of the new wall has to be occluded be the previous wall, and *vice versa*. This graphic element strengthens the impression of isometric view (fig. 4). Compared to previous problems, this issue could be solved relatively easy by inserting the newly created wall to the list of submorphs in front of or behind the previously build wall, depending on which way the bike has turned.

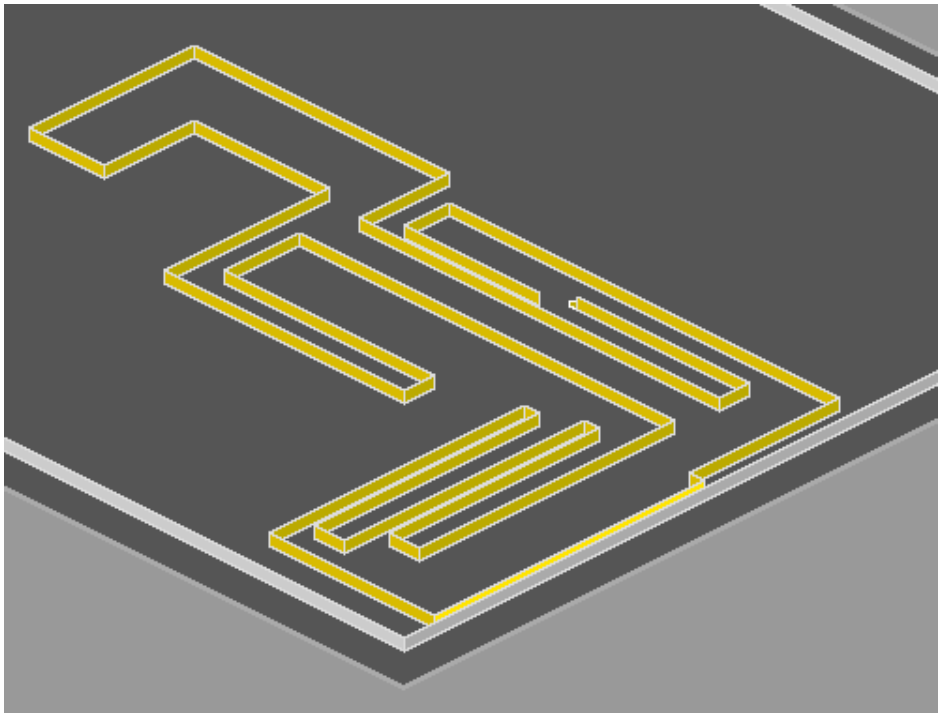

Figure 4: Results of the correct simplified Painter's algorithm. The results are good occlusion in the corners and while riding close to the outer wall. It is not, however, possible to ride so close to another light cycle wall, so that one of the walls would become occluded.

If the walls were not suppose to be drawn close to each other, there was no place for the *turbo while close to other wall* feature anymore. An exception for that are the outer grid walls, where it is easy to assume, that the two closest walls can appear always on front of any other wall, and the two last walls — always behind. Thus it is now possible in the game to run very close to an outer wall, see the partially occluded bike automatically switch it's turbo on and throw some sparkles into the air.

This last feature changes a bit the set of good strategies in the game: it no longer makes sense to try to push the opponent closer and closer to the wall with each turn, while at the wall he will simply be too fast. Now some smarter techniques have to be invented.

4.4 Other problems

As new-comers to smalltalk we had frequent problems with the order, in which squeak evaluates complex expressions. According to squeak $1 + 2 * 2$

is equal to 6, we would say it is 5.

Games of this size and complexity already have performance issues, even on quite fast machines, which in our case caused the wall occlusion problem. From time to time it was forcing us also to make small optimisation fixes, which in result probably has led to wasting a lot of time.

As a side note: we have noticed, that recent distributions of squeak for win and mac differ in encoding keyUp events. If you encounter an input problem with our game (this shouldn't occur anymore) or with somebody else's game developed under win (likely), this may be the cause.

5 Testing and playability

Author: Jakub

As already stated, we have been (obviously) testing the game during whole process of development. We needed, however, somebody else's perspective to assess, if the game was really playable.

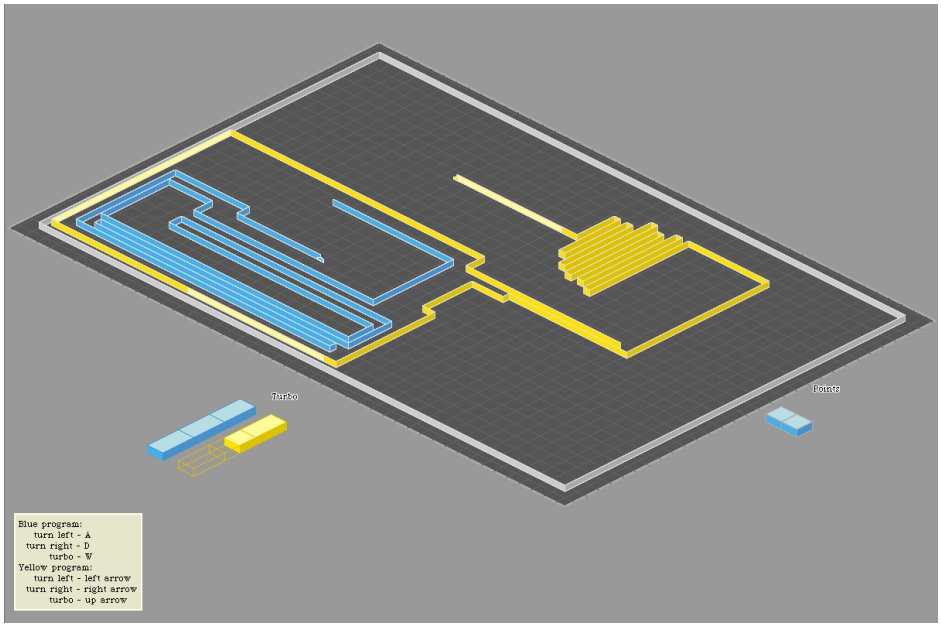

Figure 5: Final game screenshot.

Therefore we asked two of our friends, Christian and Dres, who knew little about computer science, but enough about gaming, to test our game, when it was in the last week of it's development.

The main result of their comments was changing the number of turbos from two to three per race, but shortening their burn time. Besides that they said the game was fun for a short arcade game.

Regrettably, launching the game is not very user-friendly. It is possible to write a script, which would launch Squeak with provided image, but this solution is not as comfortable, as would be deploying the game as a single file application.

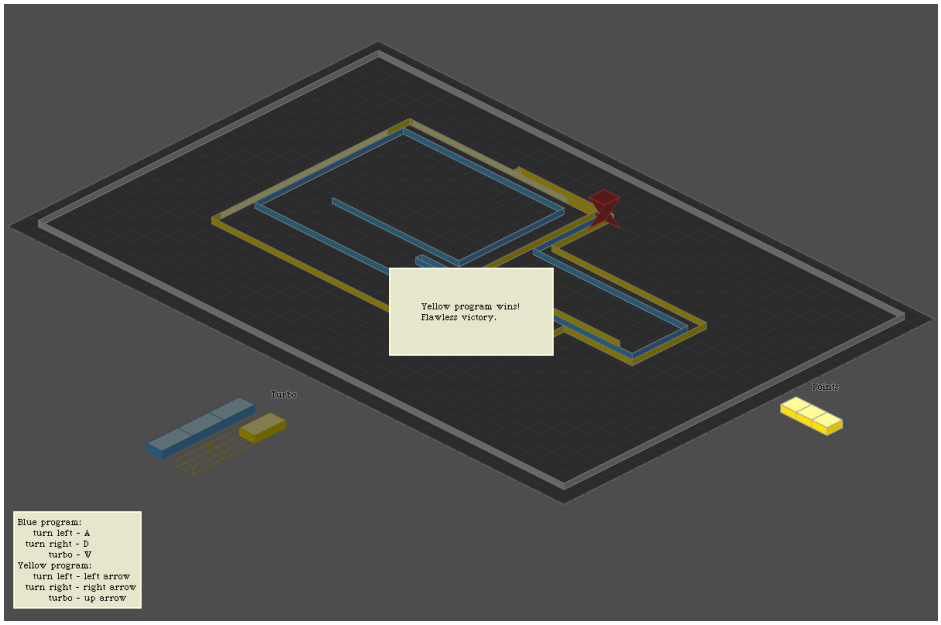

Figure 6: Final game screenshot. Yellow pro... player wins, without loosing any race.

6 Possible extensions of the game

The first extension to the game, if we were to implement it on a different platform, would be drawing the walls, like it was initially planned.

Beside that, game could sometimes be more interesting, if it had an AI module implemented. It would be not only useful for the single player mode by adding second bike, but also add additional bikes in multiplayer mode. The AI agents would have to be reactive as well as proactive — planning their strategy against the player and maybe against other agents. The best impression on a player would make a pair of agents actively chasing the player or building traps in mutual effort. This type of AI behaviour is implemented in the game *Light Cycles: Game Grid Champions*.

Like mentioned in the previous paragraph, adding more players to the competition would be interesting. It can be achieved by simply throwing

more players to the grid (real or AI), like it was done in *Armagetron*, or by organizing championships composed of two-player races, that would eliminate the weakest and lead to final duel.

Other AI-type modification to the game would use a module analysing situation on the grid after the race and displaying the clever summaries, like: *player 1 boxed himself!* or *player 1 pwned by joint effort of players 2 and 3.*

7 Assessment of Squeak as a prototyping tool

Author: Robert

Squeak is very powerful as a prototyping tool. Virtually every it's feature works in favour of that characteristic.

Squeak uses the Smalltalk programming language, which is object oriented, dynamically typed and reflective.

Smalltalk's simple syntax and dynamic typing allow for fast creation of short but functional pieces of code. In Squeak, those small pieces of code can be immediately tried out using the workspace window. A programmer writes *quick-and-dirty* implementation of his small ideas, and instantly sees the results in the environment.

Smalltalk's syntax for class inheritance, method overriding and general message-passing approach is also quite programmer-friendly. It allows for quick extending of functionality of already existing classes and refactoring previously created class structure.

The ultimate feature for a prototyping is reflection. Prototyping is all about quickly trying out many possibilities, as the program structure is not so clearly defined *a priori*. Squeak (and Smalltalk) supports that with it's reflectiveness, meaning that (almost) any changes made to the code give an immediate result in the program execution.

A huge aid in developing visual applications is Squeak's Morphic component (described in [1]). *Out-of-the-box* Morphic covers many issues the program would have to deal with, as long as objects, which have a visual representation, are concerned, i.e.: drawing, hierarchical grouping, transforming, animation, etc.

All those features lead to creating a workflow, which is based on experimenting with ideas. New morph is created, modified and displayed using workspace. Then, if some functionality has been approved during the test, it is quickly incorporated into the application's class structure. Any further fixes and fine-tuning can be made without relaunching the program.

When more than one programmer is working on a project, changes to the code can be updated quite easily by exchanging the so-called monticello packages.

However, one of the elements of Squeak we did not like, was debugging.

When an execution error occurs, debug window pops up, a stack of messages can be traced, the error identified and possibly corrected from within the same window. An exception to that is when application generates the error many times, and the environment is flooded with error messages, which is hard to stop.

Squeak provides useful tool for debugging morphs, by letting the programmer explore attributes of the current morph and it's submorphs in a tree-like view. Regrettably the window will not refresh itself dynamically, and opening it each time and clicking out the desired information, makes it useless in some situations.

The hardest aspect of debugging was tracing the program execution. Tools provided by Squeak were not very efficient, and caused us problems.

In some programs debugging can be largely avoided by using the unit-testing capabilities of Squeak.

The last thing to mention are squeak's performance issues. Squeak's virtual machine interprets and executes code *on the fly*, which probably can not be performed faster. This regrettably leads to limiting Squeak's usefulness, as long as large applications are concerned. This will most likely change over time, when faster machines become available.

Overall, we believe, that Squeak is *The* prototyping tool for small and medium-size visual applications. We look forward to seeing it further development.

8 Conclusion

Author: Jakub

During development of the project, we have achieved most of the initially defined goals. It led us to creating quite playable game, a Tron game clone.

While experimenting and implementing the game, we have gained confidence in using the Squeak development environment. It allowed us to critically (but positively) assess the tool as a software prototyping tool, as well as the prototyping approach itself.

A Division of the work

As it was stated before, we have been using the pair programming approach, therefore most of the game was created by our joint effort.

If we are to divide the responsibilities, Jakub's work was mainly to assure the proper game logic, such as control, collisions, etc. and Robert was

responsible for playability, game organisation and the graphic aspect, which included the wall occlusion problem; mostly, however, our responsibilities overlapped.

Each part of the report is marked with the name of it's main author.

References

[1] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, and Damien Pollet. *Squeak by example*. Square Bracket Associates, 2007. URL: `http://www.iam.unibe.ch/~scg/SBE/index.html` [last visited on 15.12.2007].

[2] Stephan B. Wessels. A development example for squeak 3.9 [online]. 2007 [last visited on 15.12.2007]. URL: `http://squeak.preeminent.org/tut2007/html/`.