TDT4715 - Master Project Report
Parallelization of an open source game

Idar Borlaug
Knut Imar Hagen

Main Advisor:
Dr. Anne Cathrine Elster

Fall 2006

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Abstract

Multithreading of games has recently become a hot topic. More CPUs are now being delivered with several cores. Games can greatly benefit from this. You now have more processing power to create better Artificial Intelligence, to use more physics to create more realistic games and larger and more complex games. It is a must for gaming companies to take advantage of this, because it will give you an edge compared to competitors. The goal of this project is to see how well a typical game can be parallelized on modern multicore architectures. For this task, an open source game is chosen. With an open source game it is easier to test modifications and give something back to the community. The open source game OpenTTD (Open Transport Tycoon Deluxe) is chosen as a test case we found, because the code is easy to read, and it is reasonably well documented.

This report explains how the code is analyzed and time consuming parts identified. Time consuming parts are then parallelized if possible. Game logic, sound system and graphics are modules that would increase performance if parallelized. The game logic showed to impose a few obstacles, which make it very difficult to parallelize.

The sound system is parallelized, resulting in no game play delays if the sound system waits for I/O resources. The graphics module consists of two parts: drawing the graphics, and pushing data to the graphics card. The push process is changed so that it runs in parallel with the game logic. These changes resulted in a significant speedup of 26% without significantly impacting single core performance.

This work hence illustrates how multithreaded games can take advantage of modern multicore architectures.

Preface

This report describes how to approach multithreading of an open source game, and how this is done with OpenTTD [1W].

This report is written as part of a Master Project at the Department of Computer Science at NTNU. The course name is TDT4715 - Algorithm construction and visualization, depth study. The main advisor is Dr. Anne Cathrine Elster.

The references in this report is numbered with [#X] where # is number of cited article/book, and X is W or B, which means Web or Book.

We would like to thank Dr. Anne Cathrine Elster for giving us valuable input throughout the project, and Thorvald Natvig for being a helping hand with Posix threads.

The final source code including our implementations can be found on the enclosed CD in this report, or (until summer 2007) in the subversion repository at the following address:
`http://linux-warer.homeip.net/svn`

Trondheim, December 18, 2006.

Idar Borlaug Knut Imar Hagen

Contents

List of Figures

List of Tables

1 Introduction

The goal of this project is to see how well a typical game can be parallelized on modern multicore architectures. This report investigates how to do multithreading of an already existing open source game. An open source game is a computer game that is developed by various people, often from different countries and places. The code source is free for all to download and experiment with, and anyone can play the game.

1.1 Choice of game

We decided on an open source game, because it has available source code, and it is possible to test modifications and give something back to the community. The open source game OpenTTD (Open Transport Tycoon Deluxe) [1W] was chosen because it was easy to read the code, and it was reasonably well documented, compared to other open source games we investigated. Another reason is that it is based on the well known and legendary game TTD [2W].

1.2 Game description

OpenTTD is a clone of the Microprose game "Transport Tycoon Deluxe" [2W], a popular game originally written by Chris Sawyer. It attempts to mimic the original game as closely as possible while extending it with new features. OpenTTD is licensed under the GNU General Public License version 2.0. [3W]

When you start the game, you are loaned a large amount of money to build a transporting network. The quest is to build the best network to become the richest transport tycoon in the world. You will compete against other players in a network game, or against AI players. The clue is to build the network quickest and be the smartest, so you will earn the most money. These networks consist of train and bus stations, airports, docks, linking roads, railways, air and ship networks. You also need to build all the vehicles, as trains with wagons, trucks, buses, airplanes and ships. These will transport passengers, goods, oil, lumber, iron, steel, etc. between the stations, using the transporting network. A nice part of a network is shown in a screen shot at Figure 1. As the years go by, you'll have the chance to buy more advanced and faster vehicles and ships, if you can afford them.

The networks you are building can be very complex, and you got to have good control when you connect railways in junctions. A pretty example of how things can be done is shown in a screen shot at Figure 2. You also have to deal with town councils that express individual and varying attitudes. There are also some disasters you have to handle, such as UFOs landing on your property, mine collapses, bus, lorry and aircraft malfunctions. To add more money to your wallet, you may also takeover other companies, and also fund new industries which will need your transportation network.

As a summary, the game is to build a huge transport network and make money. When a big network has been built, much CPU time is needed to run the transporting vehicles, among other CPU consuming activities.

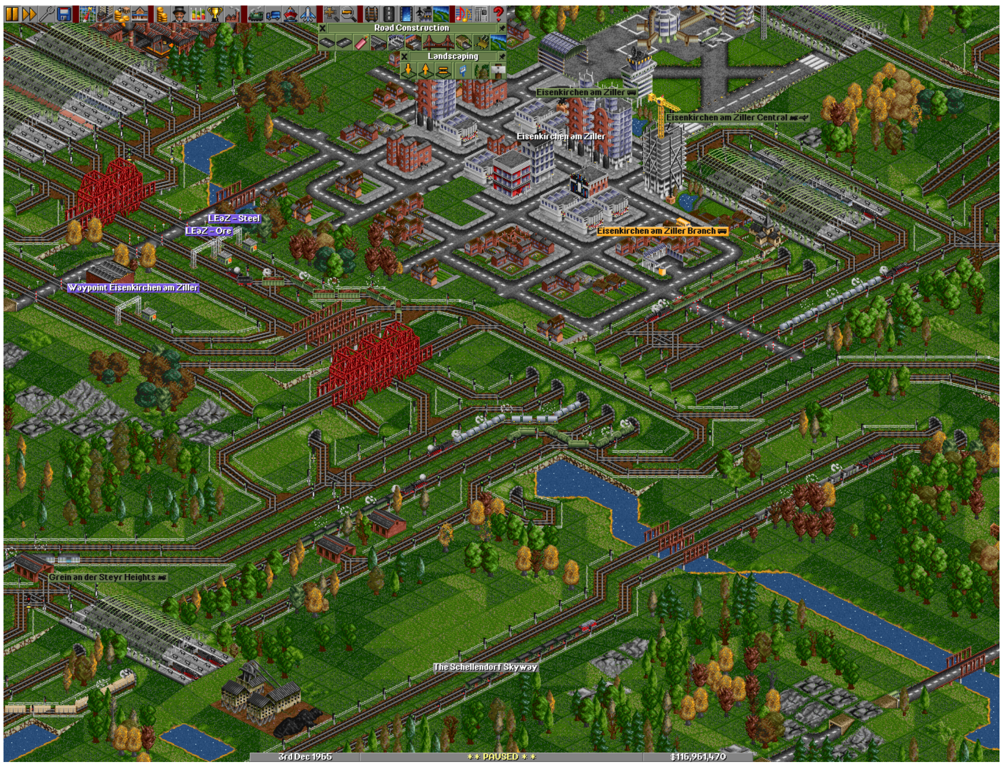

Figure 1: OpenTTD Screen shot 1

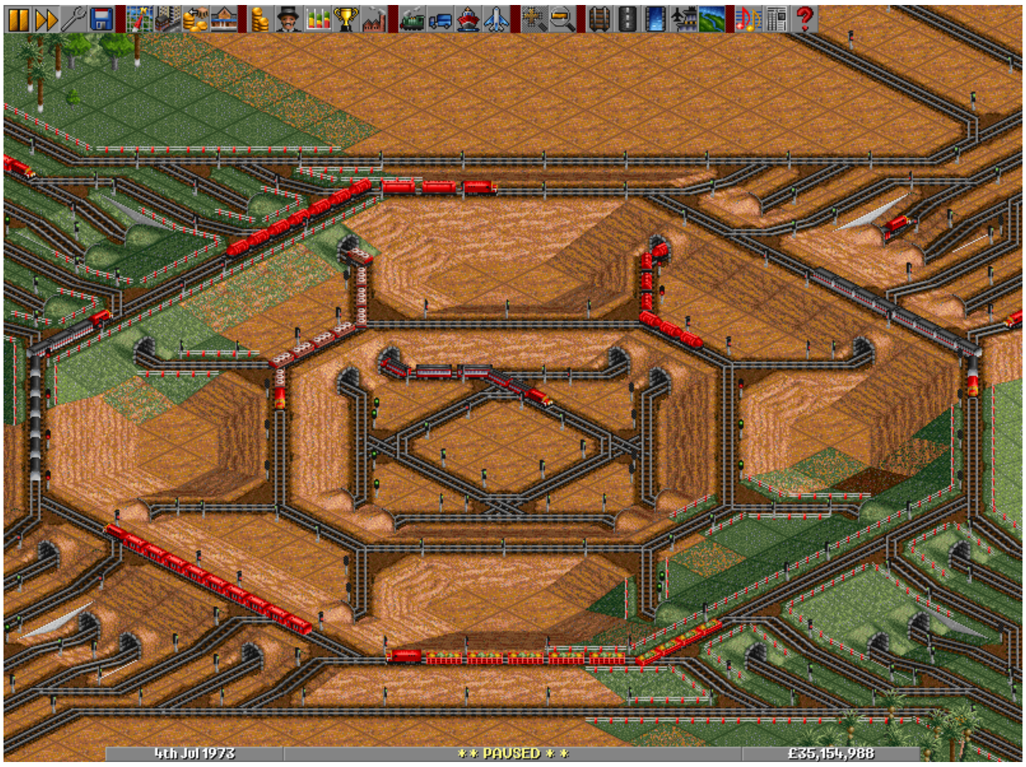

Figure 2: OpenTTD Screen shot 2

1.3 Project goal

The goal of this project is to see how well a typical game can be parallelized on modern architectures. If successful, you now have more processing power to create better Artificial Intelligence, use physics to create more realistic games, and larger and more complex games. This will give you an edge compared to competitors.

1.4 Outline

This document is organized as follows:

- Introduction where we describe the game that is chosen
- Related work in multithreading of games
- Code analysis where we have analysed the code of OpenTTD and measured time of critical portions
- Project results where we present what we have investigated and implemented
- Discussion of the project results
- Future work is a description of how OpenTTD should have been designed from the ground up if it were to support multithreading
- Conclusion of the project results
- A list of references used in this document
- A list of special abbreviations and terms used in this text

2 Related work

In the past, there has been a minor focus on multithreading games. The reason is because the games mostly are run on desktop computers, and these computers traditionally have one CPU. Recently, it has become more common to have a multicore CPU on desktop computers. A current example is Intel Core 2 Duo [4W]. This CPU has two cores, which share up to 4 MB L2 cache. Intel claims that this among other technical details yields truly parallel computing.

Nowadays, the focus on multithreading games has grown, mostly because multicore processors have become more common, as mentioned. Another reason is that the 3D graphics and Artificial Intelligence (AI) in the new games consumes more resources than the older games. A need for more resources has hence become an issue. One method is to try to optimize the already written code, another is to multithread the code. The optimal solution would be to have optimized multithreaded code. Examples of new game engines that use this technique are Doom 3 [5W] and Unreal 3 [6W]. Anandtech asked Tim Sweeney, a developer of the Unreal 3 engine about multithreading, which gives us a very good hint about multithreading [7W]:

> For multithreading optimizations, we're focusing on physics, animation updates, the renderer's scene traversal loop, sound updates, and content streaming. We are not attempting to multithread systems that are highly sequential and object-oriented, such as the gameplay.

Galactic Civilization 2 uses multithreaded AI. Since it is a turn based strategy game, it has resources available when you make your turn. These resources are used on the AI, so the AI is almost finished with its turn when you end yours. If the game wasn't developed this way, the end turn time would be long and gameplay would suffer greatly [8W] [9W].

A good example of how to improve the graphics using multithreading is described in the article "Multithreaded terrain smoothing" [10W]. Here it is described how to use multithreading to realize real-time terrain height field smoothing. Two interesting techniques that are described, are multithreaded tessellation with OpenMP, and asynchronous multithreaded tessellation. The OpenMP version seems well understandable and structured, they even apply a receipt of how to check the actual speedup, using FPS counting. It is good to be aware of reasons why the speedup is not as high as expected. One is claimed by Amdahl's law [1B] (page 40-42) that the maximum speedup you can get is limited by the serial portion of your code. The asynchronous version states that the threads can just pick up a new tile when the other is done. The main thread checks if new tiles are available, and uses the new ones if they are.

The games that are mentioned so far are commercial games. This project concentrates on an open source game. An open source game that is multithreaded is a rare case, and we have not found such a game. The reason is probably because different volunteers work on the game, and cooperation on developing a game is less tricky if multithreading is out of the picture.

Intel has written a report [11W] on how to use Intel tools on the most common open source game engines to get the best possible performance out of these. They conclude at last that dividing the source code in multiple threads will quicken execution time, if there are available processors. They have used OpenMP [12W] for parallelization.

More studies have been done by Intel on multithreading of games [13W]. A recommendation here is to use a thread pool, which is a queue of asynchronous threads to be executed. It is stated that this technique is already used in several games. An idea is to divide problems in subproblems, that is subthreads, so that they become easier to solve. In the end the solutions can be combined to a whole solution of the problem, but this is often difficult. Anyway, this is a mathematical way of thinking, that is being more and more welcomed in the game development. A final tip that is worth mentioning is to be creative and cheat to limit the need of synchronized execution of threads if the user of the game will not notice it.

As stated in the above, today most games that use threading, have parallelized parts of the game that is easily separated from the rest of the code, like particle generators, AI, sound updates, etc. A reason for this is that the game will also run normally on a single CPU. Because you will only loose visual effects, the game will work just as good. There are areas where data parallelization also can be used. Pathfinding of different units can be done at the same time, since it only reads values from the map. AI players can run independently of each other.

In multithreading, it's normal and often necessary to have locks. This is a problem because games have a high FPS. If a game has 100 FPS that means everything needs to be calculated in under 10ms for every frame. Even worse, some games have FPS up to 200, which means we have even less time. Therefore, locks and synchronization must be kept to a minimum, one each frame is a good number. Unfortunately, all problems can't be solved without locks.

3 Multithreaded game engine architectures

There are basically two ways to multithread a program / game; data and task parallelization. Data parallelization is used when there is a large dataset that needs to be processed, mostly matrix operations. This is used in scientific computing and other fields. In games, task parallelization is more common, because there are lots of different jobs (physics updates, AI, game logic, graphics, network, sound). Many of these can be run in parallel if there are few interactions between the respective parts [14W].

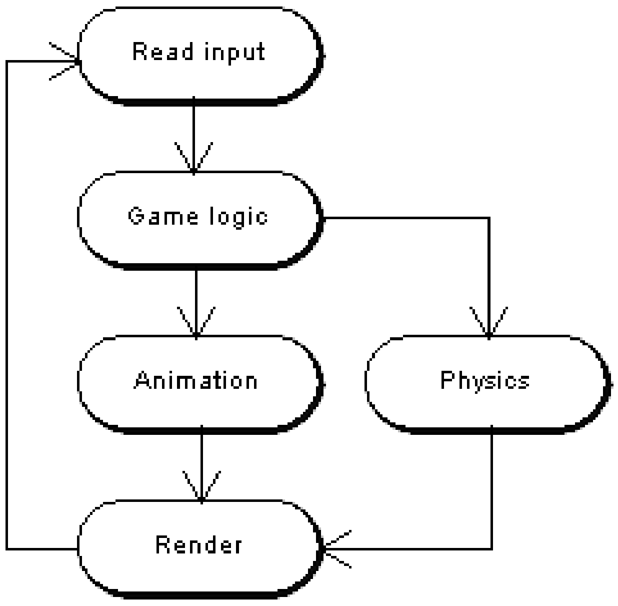

Figure 3: Synchronous function parallel model

3.1 Synchronous function parallel model

If you can find parallel tasks that can be run simultaneously from an existing game loop, they can be run in separate threads. These tasks should not interact or communicate, they should be independent. An example of this could be executing a physics task while calculating an animation. Figure 3 shows a game loop parallelized using this technique.

If you only have two threads, this will not scale well, for example on a quad core processor. Then two of the cores would have spent their time idle. This means that this model has an upper limit of how many cores it supports, the number of parallel tasks in the loop.

3.2 Asynchronous function parallel model

The last model was a synchronous version, an asynchronous one will not contain a game loop. Instead the tasks run at their own pace, and uses the latest updates available from the other tasks. You can then parallelize tasks that are dependent on each other. The information in the communication between them is gathered in an update that the other tasks use when they are ready

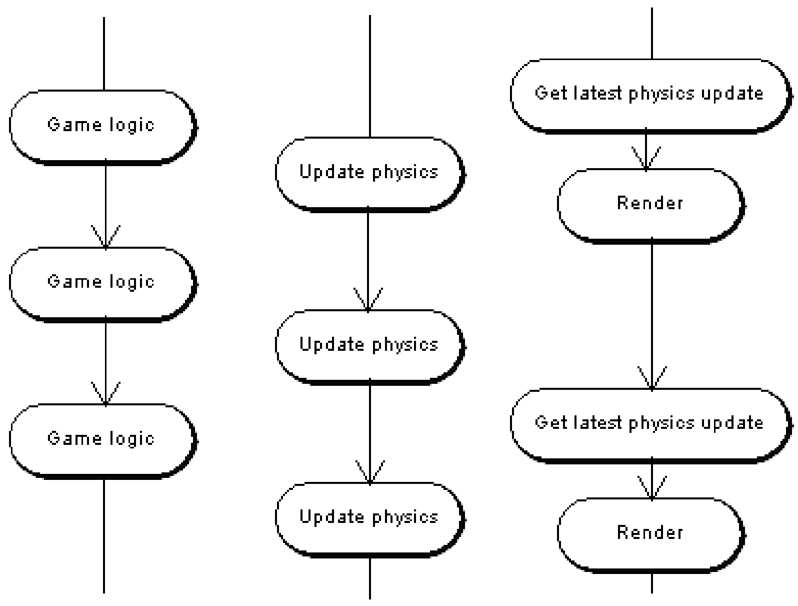

Figure 4: Asynchronous function parallel model

(polling for new data). Figure 4 shows an example of the asynchronous function parallel model.

Also this model has limited scalability, because of the number of tasks that run simultaneous. Anyway, since tasks don't need to be truly independent, this model can support a larger amount of tasks and hereby a larger amount of processor cores than the synchronous model. There are also some drawbacks. One is that if three tasks are dependent on each other, and a task that is dependent on the first starts right before the first is finished, and the third starts right before the second task is finished. This will result in a delay of up to two times the optimal scenario. Another drawback is, if the tasks use greatly different CPU time, the slowest will reduce the utilization dramatically, because the others will have to wait for it to finish.

3.3 Data parallel model

The Data parallel model is about finding similar data so the tasks on these can be performed in parallel. One example is in a first person shooter game with AI bots, such that all the bots computing can be divided in two or more threads, because they will have the same goal, to hit your character, and not each other. An optimal solution is to have as much threads as there are processor cores available. Figure 5 shows an example of the data parallel model, which have two object threads running in parallel. This means that it is suitable for a dual core, if these threads are well balanced.

The example with the AI bots assumed that the bots did not interact with each other. When a bot sees an enemy, they will need to broadcast this to the other bots, resulting in an interaction between different threads. This could reduce the amount of parallelization, because of synchronization primitives that block. Anyway, there is a solution in message passing and using the latest update available. It also helps to group objects which have the highest chance

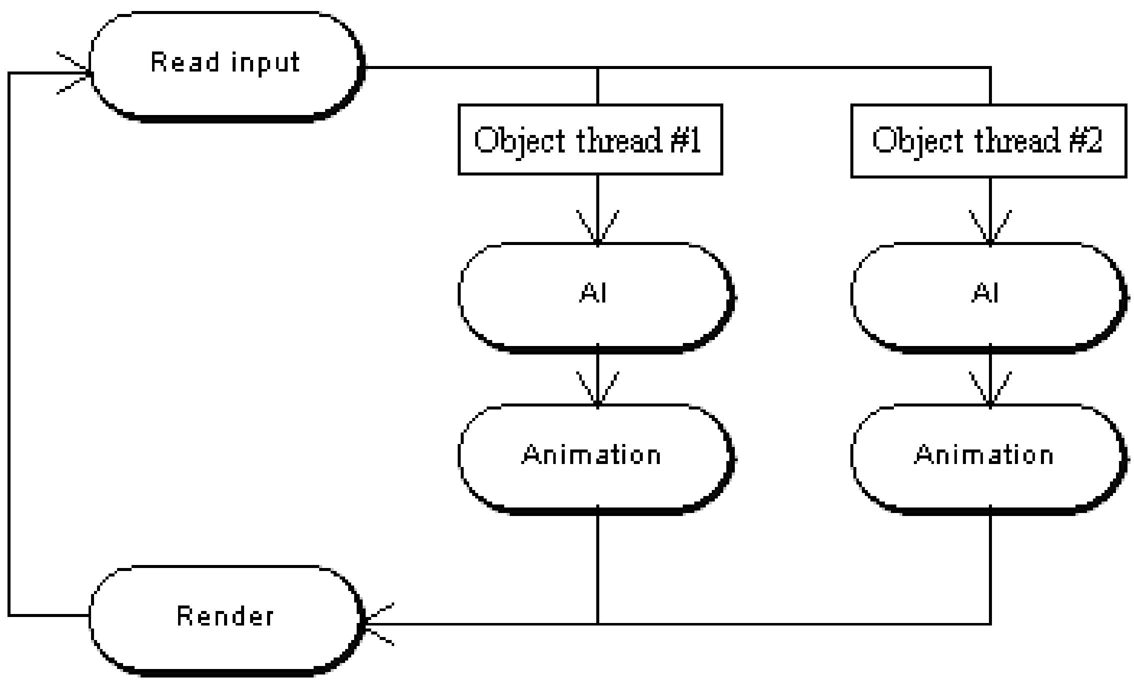

Figure 5: Data parallel model

to interact with each other. The AI bots which are far apart in the game world can be in their own threads, because they will most likely have no interaction. There are no reason for a bot to yell to another bot 5 miles away.

A great thing with this model is the excellent scalability in that it can set the number of object threads to the number of processor cores.

4 Code analysis

4.1 The code

OpenTTD is an open source game released under the GPL [3W]. The code is written in C, the ANSI-C flavor used by MS Visual Studio. Lately contributions have been accepted in C++. The code is written for cross platform compatibility and runs on all modern flavours of Unix, Win32, BeOS, OS/2 and MacOS X. The game uses SDL [15W] for graphics and input on some of the operating systems. This ensures less platform specific code.

4.2 Code layout

4.2.1 Main method

The main method takes care of initialization before it starts the main game loop. When the game loop ends, it cleans up resources and quits gracefully. The overall layout is described in Section 4.2.5.

4.2.2 The MainLoop()

The MainLoop() is rather simple. First it polls SDL for input from input devices, like mouse and keyboard. Then it sets the new tick, and calls the GameLoop(). The GameLoop() will move everything one tick forward and make changed pieces of the screen dirty. Finally, UpdateWindows() draws the dirty pieces to the screen array. Then, DrawSurfaceToScreen() sends the dirty parts to the graphics card.

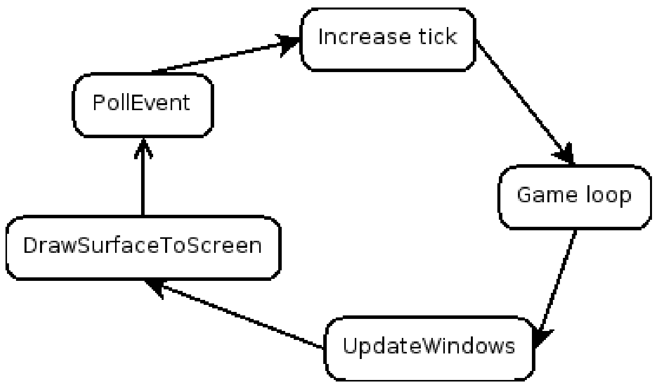

Figure 6: MainLoop()

PollEvent() is a small loop in itself, it runs through every key and mouse action taken since last loop. Every event is mapped onto a set of global variables. _cursor specifies the state of the cursor. There are variables for different mouse buttons and one for last key pressed.

IncreaseTick() will increase the global tick of the game. This will make the game logic move forward.

GameLoop() updates everything in the game to the new tick. This involves running the AI, moving all units, and animating animations. This will be discussed in a later section.

UpdateWindows() draws dirty blocks, text messages, the cursor and the landscape. This is done by looping through changes made in the game loop.

DrawSurfaceToScreen() takes all dirty rectangles and pushes them to the graphics card.

4.2.3 The GameLoop()

The GameLoop() consists of three stages, first it calls the StateGameLoop() which handles the game state and moves it forward. Second, it calls InputLoop(), which handles input from mouse/keyboard made available by PollEvent() loop earlier. Last, it calls the MusicLoop(), which makes the music play.

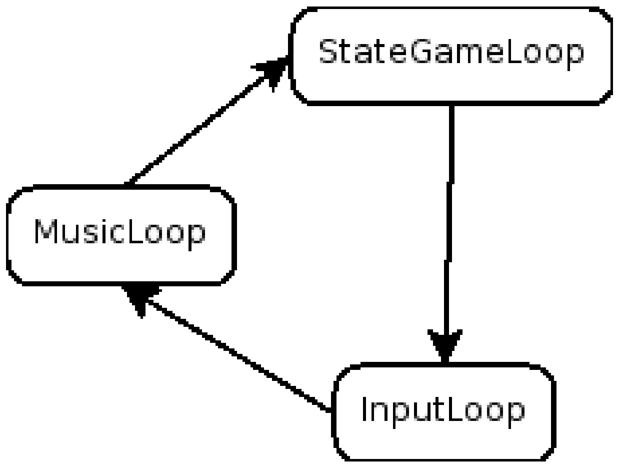

Figure 7: GameLoop()

StateGameLoop() handles all game logic and runs the AI. More info on this later.

The InputLoop() takes the variables made by the PollEvent() and modifies the game according to input; mouse movement, wheel, scrolling, clicking and keyboard interaction. The keyboard input is given to the focused window in the game.

The MusicLoop() plays the background music. It checks to see if music is playing. If it is not, it will start the new song in the play list.

4.2.4 The StateGameLoop()

StateGameLoop() takes care of all the game logic. It animates the tiles, increases the date, runs tile procs, updates all vehicles, updates landscape, runs the AI players, updates events in windows and handles news.

AnimateTile() goes through all tiles that should be animated and animates them.

IncreaseDate() increases the date of the game, and it also goes through TextMessageDailyLoop(), printing new text messages. The next loop in IncreaseDate() is DisasterDailyLoop() which checks if a disaster is ready to be launched. The last loop is WaypointDailyLoop() which runs through all way points and deletes those marked for deletion. It also runs through various Monthly and Yearly loops for players, engines, towns, industries, stations,

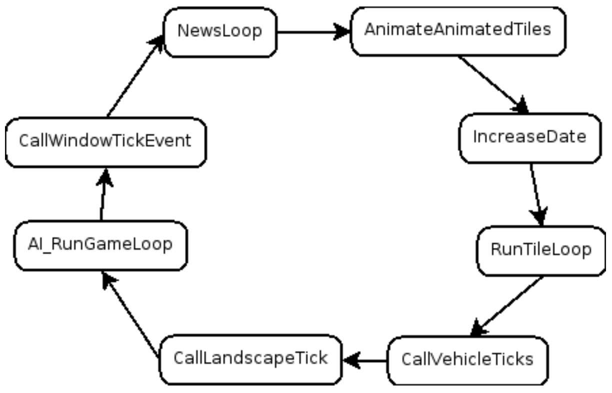

Figure 8: StateGameloop()

road vehicles, trains, aircrafts and ships. Yearly and monthly loops mostly check if the vehicles had a negative profit the last month/year, or check if they are too old, etc. IncreaseDate() also changes the last service date on all vehicles.

RunTileLoop() goes through some tiles each round and updates their status to the new tick. If the loop encounters a tile on the same level as water, and it's also next to water, the tiles are made water. On railroad tiles it will remove barren or build fences if the ground is grass. Roads are made barren, and if close to a town with road reconstruction, it will handle the road reconstruction. Station tiles only adds the animated tiles to the animation list. Town tiles will build bigger buildings, remove buildings, and move goods from houses to stations. Tree tiles add more trees or grow them to bigger trees. Bridge/-tunnel tiles doesn't do much, it only changes the ground type to the correct one. Unmoveable tiles (only a player's Head Quarter) will move cargo to stations. Industry tiles will move goods to stations, move the construction one step ahead, play sound and animate if there is an animation. Tile dummy does nothing. Clear tile removes everything on the tile.

CallVehicleticks() goes through every vehicle in the game, and calls their tick method. Finally, it goes through all vehicles in depots, for automatic replacing. The tick methods for all the vehicles are described next, and those are: TrainTick(), RoadVeh_Tick(), Ship_Tick(), Aircraft_Tick(), EffectVehicleTick() and DisasterVehicle_Tick().

TrainTick() will increase the tick of the train by one. It will also Increase the number of days that cargo has been in transit. Next, it handles crash animation, reversing of direction, stopping the train, loading and unloading of cargo, smoke animation, check if we collided, setting speed, handle the orders for the train and choose which direction the train should take.

RoadVeh_Tick() will increase tick and cargo days. It will check if it has crashed with a train and handle the corresponding animation. It will check the orders and handle loading of goods. If vehicle is in a depot, it will be serviced and moved out. Speed is set, and the vehicles are moved forward on the road to their new position.

Ship_Tick() will increase tick and cargo days, handle ship orders, loading and unloading of cargo, depot handling and find new direction and move the ship towards that direction.

Aircraft_Tick() will increase tick and cargo days, animate crash animation, handle breakdowns, smoke animation, process orders, loading and unloading of cargo and move the aircraft to the next position.

EffectVehicleTick() will handle the animations of chimney smoke, steam smoke, diesel smoke, electric spark, explosions, normal smoke, breakdown smoke, bulldozer and bubble.

DisasterVehicle_Tick() will handle the moving of animations and disasters.

CallLandscapeTick() grows the towns, which means making new and larger buildings, builds roads and changes terrain to fit the town. It plants a new tree at a random spot. All stations will update their ratings and new acceptance is set. Industries will produce goods. It goes through players and checks if they have a company name, then generates one if they don't, maybe starts an AI player.

AI_RunGameLoop() runs the AI and decides on which commands the AI should take.

CallWindowTickEvent() will call an event function on all windows. This will have a different behavior on different types of windows, mostly updating window data and marking changed sections as dirty.

NewsLoop() checks if there are new news articles. If there are articles, the first will be displayed.

4.2.5 Overall design

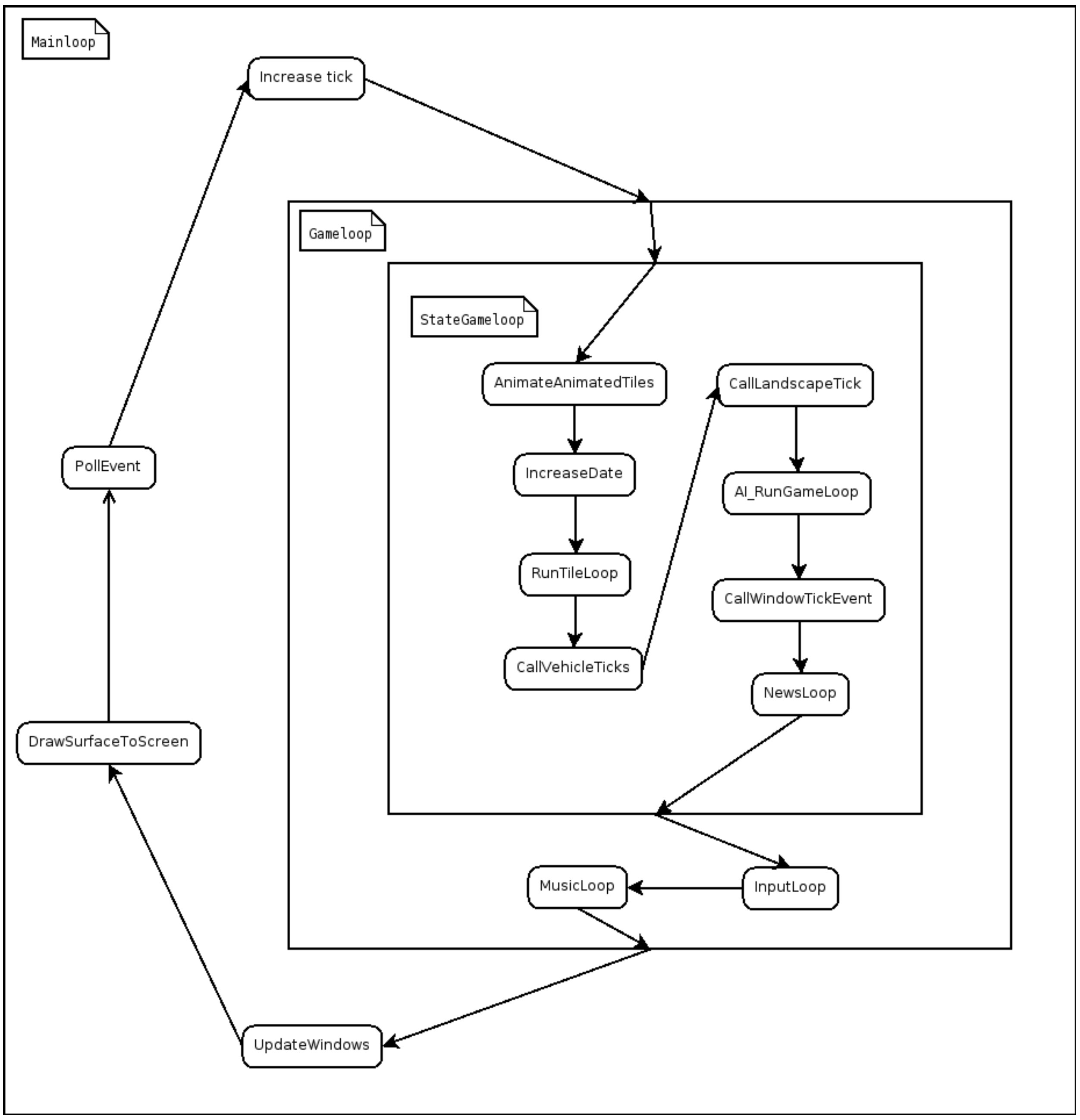

Figure 9: Overall design

4.3 Performance analysis

The measurements done here are wallclock time, averaged over several ticks when playing the game, and will not reflect every aspect of the game play, only a special case. When another player is playing, he/she will probably do it in another way and that will give completely different measurement times. These measurements are only used as a rough estimate of how much time different parts of the game code consumes.

4.3.1 Game with small map and fewer vehicles

Function	Loops	avg time in micros
GameLoop	1246	439,58
UpdateWindows	1092	238,67
DrawSurfaceToScreen	525	6744

Table 1: Small game, GameLoop()

You can see that GameLoop() uses twice as much time as the drawing loop (UpdateWindows). DrawSurfaceToScreen() runs 10 times longer than the other two. This is not a correct number, because DrawSurfaceToScreen() uses a lot of CPU time in the loops when it has to update a lot of rectangles on the screen. When there are no rectangles left to update, it takes less than five microseconds.

Function	Loops	avg time in micros
NewsLoop	766	3,46
InputLoop	766	14,37
CallWindowsTickEvent	766	5,42
CallLandscapeTick	766	22,211
CallVehicleTicks	766	326,96
RunTileLoop	766	127,45
IncreaseDate	766	16,34
AnimateAnimatedTiles	766	6,74

Table 2: Small game, StateGameLoop()

CallVehicleTicks() is the most expensive loop within the GameLoop(), next comes RunTileLoop().

4.3.2 Larger map and more vehicles

Function	Loops	avg time in micros
GameLoop	1713	1651
UpdateWindows	1714	1077
DrawSurfaceToScreen	2233	4225

Table 3: Large game, GameLoop()

With a larger map and more vehicles, the GameLoop() has increased in time, the drawing loop also increased. It seems that the GameLoop() still is the slowest loop excluding DrawSurfaceToScreen(). As you can see, time spent in DrawSurfaceToScreen() is less here than in the small game. This reflects that DrawSurfaceToScreen() is not dependent on the size of the game, but on the display size.

Function	Loops	avg time in micros
NewsLoop	1736	3,38
InputLoop	1736	7,5
CallWindowsTickEvent	1736	5,35
CallLandscapeTick	1736	60,62
CallVehicleTicks	1736	1099,0
RunTileLoop	1736	254,47
IncreaseDate	1735	78,63

Table 4: Large game, StateGameLoop()

CallVehicleTicks() is the loop that increased the most with more vehicles and larger map. With even bigger maps and even more vehicles, this will probably become very expensive.

From the measurements, it is apparent that drawing takes a lot of time. Anyway, the game logic is still worse. It would be preferable to do some logic in the GameLoop() in parallel. CallVehicleTicks() uses so much compared to the others, that this one must at least run in its own thread. That would remove all the time from the other loops.

5 Project results

This section describes the different modules that we planned to multithread, the problems we encountered during implementation and the solutions.

We made a framework for encapsulating threads [16W] in the game. The reason we did this was because of code reuse, and making the code more neat, but mostly because Posix threads [16W] are not supported by all platforms, as this game is intended to be. That means that for other operating systems, the same methods have to be implemented. The methods for Unix is implemented in the file thread.c which existed in the source when we started on this project. The whole thread.c file is included in Appendix A, thread.c. The following functions have been added:

- struct OTTDMutex
- struct OTTDCondition
- struct OTTDBarrier
- OTTDMutex* OTTDMutexCreate(void)
- int OTTDMutexLock(OTTDMutex* mu)
- int OTTDMutexUnlock(OTTDMutex* mu)
- int OTTDMutexDestroy(OTTDMutex *mu)
- OTTDCondition* OTTDConditionCreate(void)
- int OTTDConditionSignal(OTTDCondition* co)
- int OTTDConditionWait(OTTDCondition* co)
- int OTTDConditionTimedWait(OTTDCondition* co, int msec)
- int OTTDConditonDestroy(OTTDCondition *co)
- OTTDBarrier* OTTDBarrierInit(int number)
- void OTTDBarrierWait(OTTDBarrier* barrier)

Functions for creating, joining and exiting threads were created before, so we used these in our code.

5.1 Sound

The sound subsystem was chosen as a good candidate to thread mainly for two reasons: it contains a decent amount of I/O, and it's not vital that the sound is 100% synchronized with the system.

5.1.1 Description

When the sound system gets a request to play a sound, it loads the sound from disk and pushes it to the sound card. This involves disk read and the moving of data to the sound card, both of which can have spikes in I/O wait. If this was put in a separate thread, the game would continue as normal, but the sound would come a few frames later. Most people wouldn't mind hearing the sound a few frames later to avoid a frame lag.

5.1.2 Solution

A worker thread for playing sound was implemented. The main game thread sends which sounds are to be played to a queue. It then wakes up the sound thread. The sound thread reads which sounds are queued, and play these in correct order until the queue is empty. It then goes back to sleep, waiting for a new awakening. Fortunately, the queue implementation which was present in the game, was already thread safe when one thread pops and one pushes.

The file-IO subsystem was modified. Since all the file accesses through this file-IO subsystem was read-only, there is no reason why we shouldn't have one file handle for each thread. We therefore created a hash map to store the different file handles in together with the positioning variable. This created another problem. The GenerateWorld thread uses some code that depends on open files, and the files being in a specific position. Since the main thread was in wait mode when the generate world thread was running, there didn't seem to be a problem. Therefore we stored the thread-id for the sound thread and used that when the sound thread wanted to access files, else we used 0. Therefore the two other threads now use the same file handles and the sound thread has one of its own.

Code is located in Appendix A, sound.c.

5.1.3 Problems

When we started the game for the first time with the sound thread, it ran fine for a while, then crashed. It appeared that the file-IO subsystem used a global positioning variable to remember the file position. This made a bit of trouble when two threads tried to seek to their respective parts of a file. The last one to seek gave the other thread invalid data.

5.1.4 Lessons learned

It is a very bad idea to keep a generic global pointer to files being read. When there are two threads that read from the same file, and both want to move the file pointer to a location they want to read from, one of the threads will get the wrong data and most likely crash in some way.

5.2 StateGameLoop()

The StateGameLoop() is located in the openttd.c file. This loop is thoroughly described in Section 4.2.3 and 4.2.4.

5.2.1 Description

The Section 4.3 Performance analysis, indicates that functions worth multi-threading are CallVehicleTicks() and RunTileLoop(), because these have the highest time value. If these functions are put in two threads, they can be run in parallel with each other, and with the other functions. The pathfinding is run from CallVehicleTicks() and it is used to compute the path where trains are going to travel along the track rail, where buses/trucks are going to drive along the road, or where ships are going to sail to avoid hitting land during the voyage. The pathfinding algorithm is rather intensive in its usage of CPU resources. Actually, when more vehicles are in the map, then more resources are used, so this is something worth parallelizing, and the worth increases as the game time goes by.

5.2.2 Solution

First, we made one thread each for the functions CallVehicleTicks() and RunTileLoop(). These were created at the top of the StateGameLoop() and joined at the bottom. With this code, they will run in parallel with each other, and the other methods in StateGameLoop(). This worked fine, but we realized that creating two threads for every time StateGameLoop() is called, is time consuming. That is because StateGameLoop() is called in every VideoMainLoop() iteration. Therefore we made an initializing method that created two threads, and used a barrier to synchronize the threads with the main thread. Now, there will not be time wasted in creating the threads. Anyway, time spent in the barrier calls are also a bit time consuming, but it is not significant (three microseconds per barrier).

Because CallVehicleTicks() is so intensive and time consuming, mostly because of the pathfinding algorithm, we needed to parallelize this method as well. This was done with a division of the loop where all vehicles in the game are investigated, and different methods are performed on these, including the pathfinding. One thread now runs a loop with all the trains, and another thread runs the remaining vehicles.

From Figure 10 we see that there are four threads, including the main thread, running at the same time in StateGameLoop().

5.2.3 Problems

A lot of trouble, which was not foreseen, appeared during this particular module. These errors were random, and it could go hours and days before they actually crashed the program. The game was tested on a single core processor.

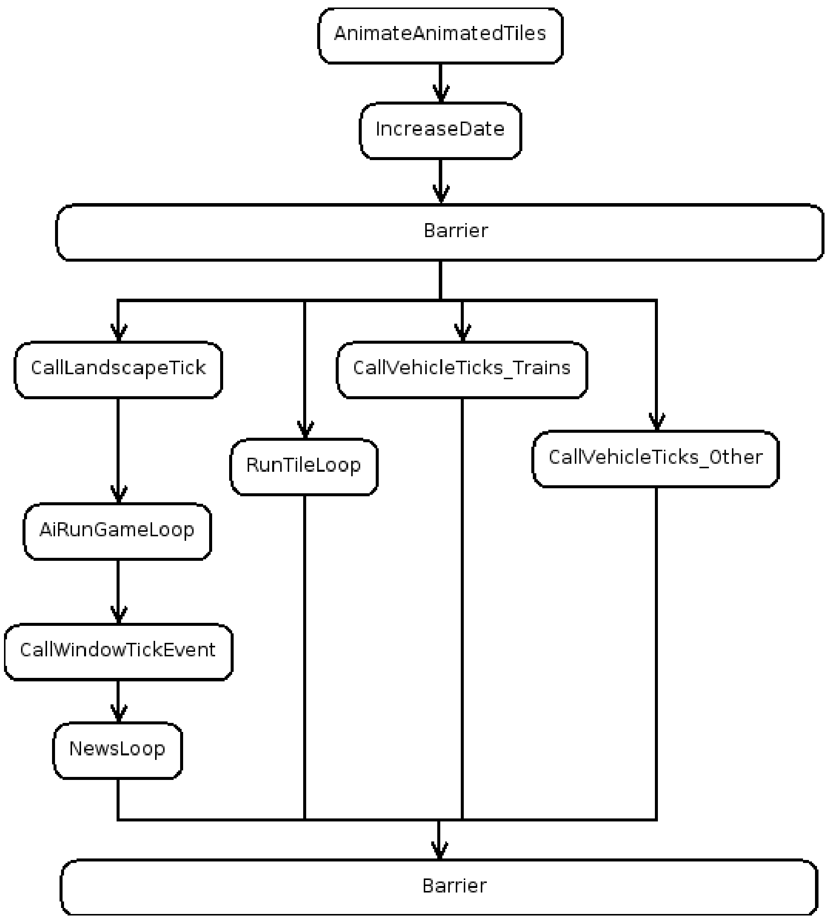

Figure 10: StateGameloop() threaded 1

LoadUnLoadVehicle() reported some problems. Apparently the gobal _players array was manipulated by different threads. It was difficult to find out where this was done. The function sets _current_player to vehicle owner, and then it calls functions that read _current_player to set some attribute on the player. This is done instead of just passing player as an argument. This is now done in our code in those methods called from LoadUnLoadVehicle(). It effectively removed problems relating to _players and _current_player.

VehiclePositionChanged() runs the method GetRawSprite(), which gave a segmentation fault. This is because several threads use that method, and images have been removed when this thread tries to access the sprite for the image reference. This was solved by locking the actual GetSprite() method in VehiclePositionChanged() with a mutex.

GetItemFromPool() is an inline function that reports an assert failure. This is something that is unexplainable, and it remains as a random error.

Another error was in the gameplay when we saw that trains which had goods wagons attached, changed the image of its wagons when it drove along a vertical track, but vertical only. This error happened only once, but it will most likely happen again in another game play.

When running this on a dual core processor, things got even worse, and new random errors appeared. Anyway, this time the errors appeared more often, which is what we expected.

The biggest problem was that the network playing did not work anymore, because it no longer was synchronized. Then, we decided to begin all over with the StateGameLoop(). The next section describes the new parallelization of this loop.

5.2.4 Lessons learned

Consider time spent in creating and joining threads vs using barrier. Don't use global variables as arguments for functions.

5.3 StateGameLoop() second attempt

As there were a lot of problems with our first attempt to parallelize the StateGameLoop(), this is our next attempt.

5.3.1 Description

Since CallVehicleTicks() gave us many problems, we took another approach. We tried to make CallVehicleTicks() run in the main thread, and export everything else to another thread. This would not be an optimal solution, but it will give a good starting point to approach CallVehicleTicks() later.

5.3.2 Solution

The solution now is to create another thread that will leviate some of the work from the main thread. We created two barriers before and after StateGameLoop(). AnimateAnimatedTiles() is left at the top together with IncreaseDate(), because when these were put into the extra thread, they created a lot of problems. They

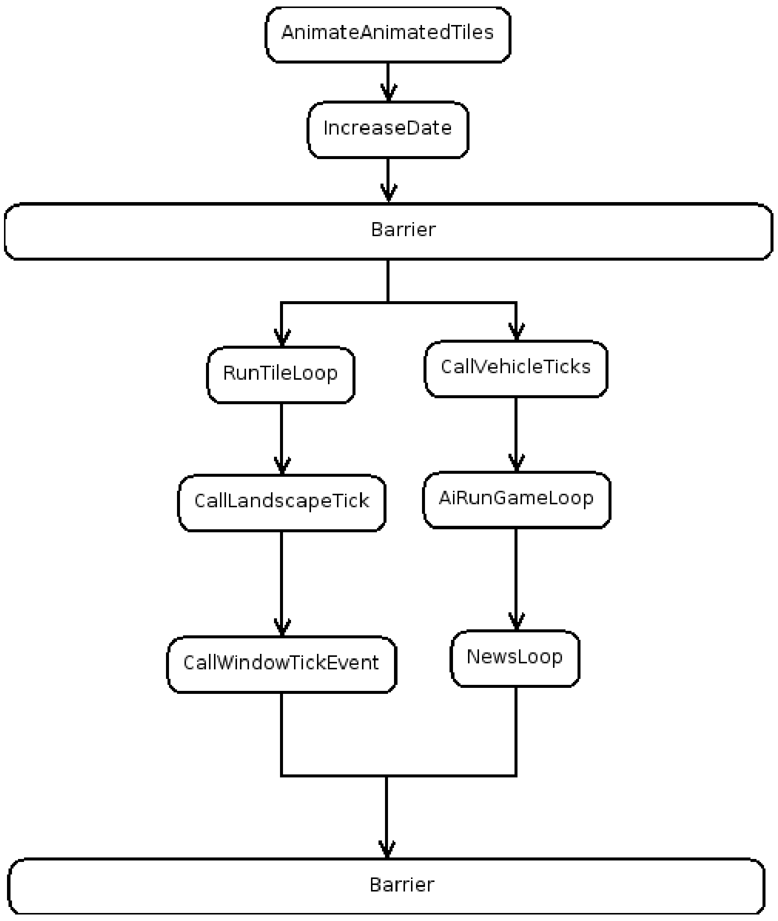

Figure 11: StateGameloop() threaded 2

are not the most intensive functions either. What we managed to leviate was RunTileLoop(), CallLandscapeTicks() and CallWindowTickEvent().

5.3.3 Problems

When parallelizing the RunTileLoop(), we got the same problem here as with our first attempt. LoadUnLoadVehicle() created problems with the global _players array. Because they used the wonderful global variables as input arguments, we created Player input arguments to all functions under LoadUnLoadVehicle() and changed all other places where these functions were used.

AnimateAnimatedTiles() when moved, created a lot of different segmentation fault errors. The same with IncreaseDate(). Apparently the program needs to run AnimateAnimatedTiles() before IncreaseDate() and RunTileLoop() after IncreaseDate().

This worked very well in single player, it ran for days without any problems. When we started multiplayer, it ran for quite some time. After a while we got a random synchronization error. This happens with the original game too, so we thought it was not a big issue. Later, when we had a larger game with more vehicles, the synchronization errors started to become more frequent and in the end they appeared after a few minutes.

5.3.4 Lessons learned

Take small steps and test thoroughly. Since this is our second attempt, we test with dual core and network play after each minor change.

5.4 StateGameLoop() third attempt

We gave a final attempt to get networking to function properly.

5.4.1 Description

Since network playing actually didn't work as well as we had hoped in the second attempt of the StateGameLoop(), we now give it another try.

5.4.2 Solution

We first started with RunTileLoop() as it was the most CPU intensive function after CallVehicleTicks(). Instead of dividing tasks, we now tried to data parallelize the loop. RunTileLoop() loops through the map array with an interval of 8 tiles. So in every tick, 1/8 of the tiles are processed. What we first tried was to divide this loop in two threads with each thread using an interval of 16 tiles starting 8 tiles from each other. This would split the loop in half. Anyway, this did not remove the synchronization problems. Therefore, we tried two loops with an interval of 8. Instead of taking all tile types in one loop, we used if statements, so some of the tile types were processed by one thread and the rest by the other thread. This worked for some tile types.

CallLandscapeTicks() calls seven different functions. We tried to do these in parallel. Three of these had to be run in sequence in one thread, so those were put in the main thread. We then put one function in the other thread, because we wanted also that thread to do something. The two remaining functions were taken by the thread that claimed to run the function first. This will help even the load between the two threads. It was implemented by a mutex for each function, and a boolean variable which said if the function was already run or not.

5.4.3 Problems

RunTileLoop() gave us some headache, increasing the interval still gave network synchronization problems. When we divided by tile type, it worked for some tile types, but not others. After a while of trying and failing, we found a few combinations of division of tile types that worked with network play. Unfortunately they were the types that did very little work. CallLandscapeTicks() worked great, but normally it uses only 20 microseconds which accounts for very little of the time each tick. We could however see that both CPUs were used during gameplay.

The multiplayer protocol for OpenTTD works by comparing the numbers received by the InteractiveRandom() function on each client. This function is called a number of times throughout the game code. If each client does exactly the same each tick, it will be the same on each client. However, when we started to multithread, things worked perfectly in single player, because in single player it wasn't that important if one action was delayed a tick because of a race condition somewhere in the code. When we tried multiplayer, it created a few problems, because when thread synchronization issues delayed something one tick, InteractiveRandom() would not be called the same amount of times on each client.

5.4.4 Lessons learned

Allowing things to be randomly executed is not a good idea with network synchronization, because it can create inconsistencies between the game state on different clients.

5.5 Graphics

The Graphics system is rather processor intensive. So much of the time in a frame is spent doing graphics calculations. This is the reason we chose to look at this system.

5.5.1 Description

When GameLoop() is finished, UpdateWindows() is executed. UpdateWindows() updates all the windows and calls methods that draw different parts of the screen. To draw the screen, it needs to know what state the game is in at this particular moment. This is stored in quite a few variables. There are lists with different vehicle types, there is an array describing the map. There is a global date variable, a tick variable, etc. So there are a lot of global variables describing game state. The GameLoop() also invalidates certain parts of the screen, where objects have moved, so that they will be redrawn.

When UpdateWindows() then comes along, it will start drawing everything that has been changed. It will go through all windows, and redraw them as necessary. This will start by calling some window events, and then DrawDirtyBlocks(). DrawDirtyBlocks() will go through the viewport (window you see) and redraw everything that is dirty, with DrawOverlappedWindowForAll(). Now, it will redraw everything that is within that rectangle, with a global pointer to where in the rectangle we are. Because many functions draw different things, they are called in a specific order and will draw their respective parts of the rectangle. This will happen in order through the use of the global pointer, which will move as they draw different parts of the rectangle.

Last, DrawSurfaceToScreen() will take the changed pixel array and push it to the graphics card.

5.5.2 Solution

Each window has its own window event which could have been parallelized. That is only if you have windows on your screen, which you normally have very few of. If you have windows, their information isn't updated every tick. However, the viewport is updated when something changes in the area where the viewport is.

We didn't change the code under UpdateWindows() because it created a lot of problems and would require a complete rewrite of the drawing functions.

We did, however, run DrawSurfaceToScreen() in another thread. This function reads from the pixel array and puts the changed data to the graphics card. The pixel array is updated through a bunch of functions in UpdateWindows(), so it couldn't go in parallel with UpdateWindows(). However, the GameLoop() doesn't modify the pixel array, it just marks tiles as dirty for the UpdateWindows() function to redraw. So now, DrawSurfaceToScreen() runs in parallel with the GameLoop() as shown in Figure 12.

Code is located in Appendix A, sdl_v.c.

5.5.3 Problems

It was a big problem with the code that all game state data was spread out on many different variables. To change this, would have required a complete rewriting of the game.

DrawDirtyBlocks() goes through everything and redraws rectangles to the pixel array. This would also be possible to parallelize, but each of the functions called under DrawDirtyBlocks() draw different parts of the screen. These functions are dependent on the global pointer to the pixel array. They move the pointer ahead when they are done drawing one area. This means that all drawing functions would need to be rewritten with pointers to the pixel array as private input parameters.

Also the sprite cache used, was not thread safe. When two threads tried to load sprites at the same time, it corrupted the cache array.

It would be possible to use two different pixel arrays and draw to each one in a round robin fashion. To get this to work, you need to draw what

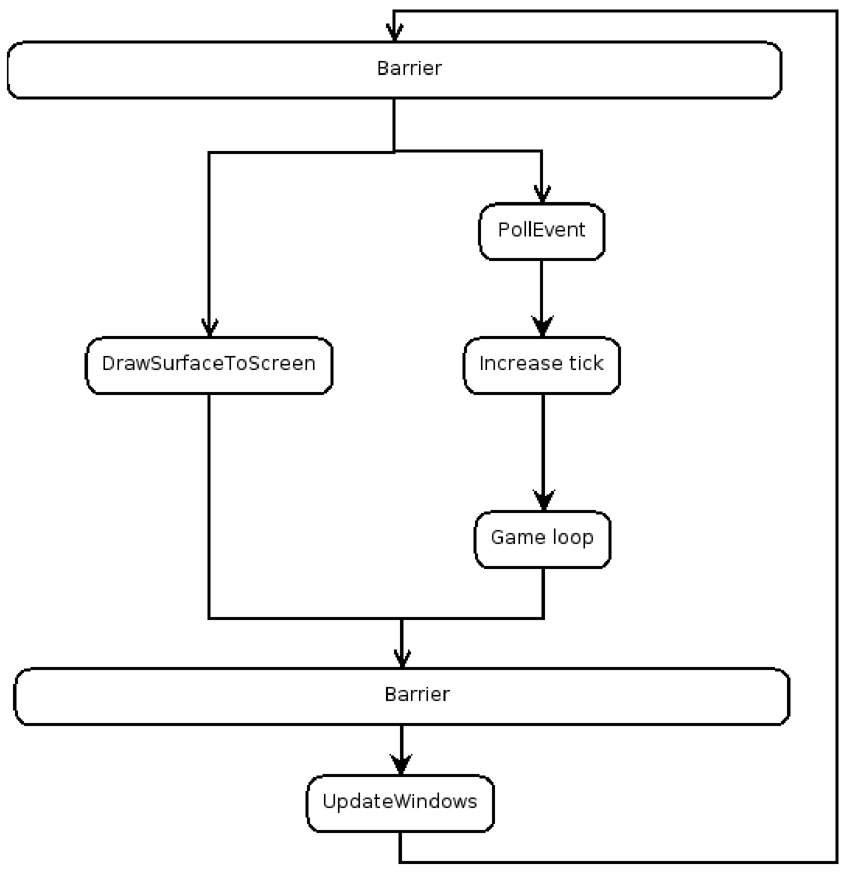

Figure 12: DrawSurfaceToScreen

was changed in the two previous ticks, so you need to store dirty tiles for two ticks. If you only paint the last tick, the pixel arrays would only get every other update. If this was implemented, UpdateWindows() would take, in worst case, twice as long to run. Anyway, DrawSurfaceToScreen() would be able to run in parallel with UpdateWindows(). As you can see, this is not favorable with only one CPU, because it would then create extra processing.

5.5.4 Lessons learned

It's important not to have two threads calling SDL. Only one thread should call SDL functions.

6 Discussion

The goal of this project was to see how well a typical game could be parallelized on modern multicore architectures. Because we didn't have time to rewrite the whole game, this needed to be done without rewriting too much code. Therefore we tried to parallelize parts of the game with minor modifications. This resulted in many problems with the code, that made it very difficult to parallelize. There were a huge amount of global variables used from many different functions, and recursive functions that counted the recursion in global variables. As a result, the outcome was not as good as we had hoped. These problems occurred when we tried to run threads within the GameLoop(). Because this code was not intended to be parallelized when it was created, many years ago, it required a lot of work. We changed a few functions so they did not make segmentation faults, but still there were problems. Especially with the network synchronization.

We started a bit too late with testing of the network, so we didn't discover those problems before we had done a lot of work on the game code. This resulted in a major setback. It would have been better to start testing the network gameplay right away. In single player, though, we were able to gain some significant speedup. Unfortunately it broke the network synchronization.

The sound system also got an overhaul. We changed it from being an integrated part of the GameLoop() to be a separate thread that plays sound when it has the resources. Playing of sounds involves reading data from the disk and pushing this to the sound card. Reading from disk can sometimes take much longer than one might think, and even worse if there is additional disk activity. Now, if this happens, the sound will only be delayed until the data is ready from the disk. The game will continue as nothing happened. We were happy with this solution, it removed a problem that might create infrequent lags in the game.

We were also able to parallelize the DrawSurfaceToScreen() function together with GameLoop(). If we use the average numbers obtained from our tests, it will give an increase of 6% and 23% in the small and large game we tested. Again, these numbers should not be taken literally. It's at best, a very rough estimate, because of the very high variation in the time it takes to run the two methods. Because of this, it was impossible to see if the spikes in both functions correlate. If the time correlate to some degree, it might give even better results. If we were unlucky and they didn't correlate, it would be even worse. This was only for two games, and as you can see from those numbers, the smaller game uses more time in DrawSurfaceToScreen() than the larger game. It should be roughly equal, only screen resolution should affect the time in DrawSurfaceToScreen() or the amount of dirty pieces on the screen. Now, it looks like DrawSurfaceToScreen() takes a lot longer than the GameLoop(). If there is added more complex game logic to the GameLoop() (better AI, more advanced pathfinding, etc.) it will not increase the time it takes to complete a tick on a dual CPU machine. So this will make it more compelling for the developers to create a better game.

If you design a game from the ground up, you can take a lot of considerations that would make it much easier to multithread. Separating code into separate modules and try to avoid communication between different modules. This will result in a much better performance increase than we achieved. It will be possible to parallelize parts of the code that in our case would require a total rewrite of the program.

As we saw in our early research, very few in the game industry have experience with multithreading. It will be costly to make multithreaded games, because it will require more time spent in debugging and more time spent in coding. In the future, game manufacturers won't have a choice, the CPU will contain more and more cores in the foreseeable future.

7 Future work - redesigning the game

Because of the many problems with the existing code, a future approach would be to start from scratch, and design the game from a multithreaded perspective. In this section, such a design of OpenTTD as a multithreaded game, is described.

7.1 Graphics

Since this is a 2D game, all graphics are processed on the CPU. There is a pointer to the array, defining the screen that decides where to draw. Instead of drawing to this array sequentially through the entire array, you can divide it into different discrete parts. So one thread is responsible for drawing x(0,200) and y(0,200). The next one x(200,400) and y(0,200) etc. This will make 4 threads doing their own parts with no interference from the other threads. There would be some overhead with calculating which windows should be on the respective parts of the screen. Anyway, this is also done today, it's calculated which parts are inside the screen, and which are not.

This might lead to different completion times for the different parts of the screen, and one thread might be finished before the others. To correct this, the screen could be split in 16 pieces and 4 threads could go through them one at a time. Then, the completion time would be much closer to each other.

7.2 PushGraphics

When the graphics drawer threads are done, another thread should start pushing this data to the graphics card. This can be done simultaneously with the game data processing and the reading of user input.

7.3 Sound

Sound could be put into a separate thread and played when the CPU has resources for it. Put the sound requests in a queue and play them when resources are available for it. That is exactly like we did.

7.4 User Input

User input is now handled in the beginning of the loop. This is important to finalize before you run code that depends on the input. Today, the stored input is read from different parts of the code. SDL does caching of user input, so the input is ready when your thread reads the input. This means, it's not the most CPU intensive work. We would suggest just putting it in front of game state and leave it single threaded.

7.5 Game state

Today, the game state is sequential. We have parallelized some of the code. With a redesign it's important to make the CallVehicleTicks() thread safe, because here is where most of the work is done when updating game state. Unfortunately, there are some parts of the code that need to interact with each other. It might be easiest to run a loop with trains because trains interact with other trains, but not ships and planes. A lot of work is needed to redesign this code. The problem with running one kind of vehicle, is that in most games there is a huge amount of trains and planes, which will make an uneven amount of work. Again, this time has a high deviation from frame to frame, so the best model for game state would be to have a work controller, which worker threads take tasks from in a dynamic fashion. Then, one thread could start with trains and another thread could do all the other vehicles, and continue on tile loop for updating tiles.

7.6 New design

Figure 13 is an overview of the implementation of the redesigned game.

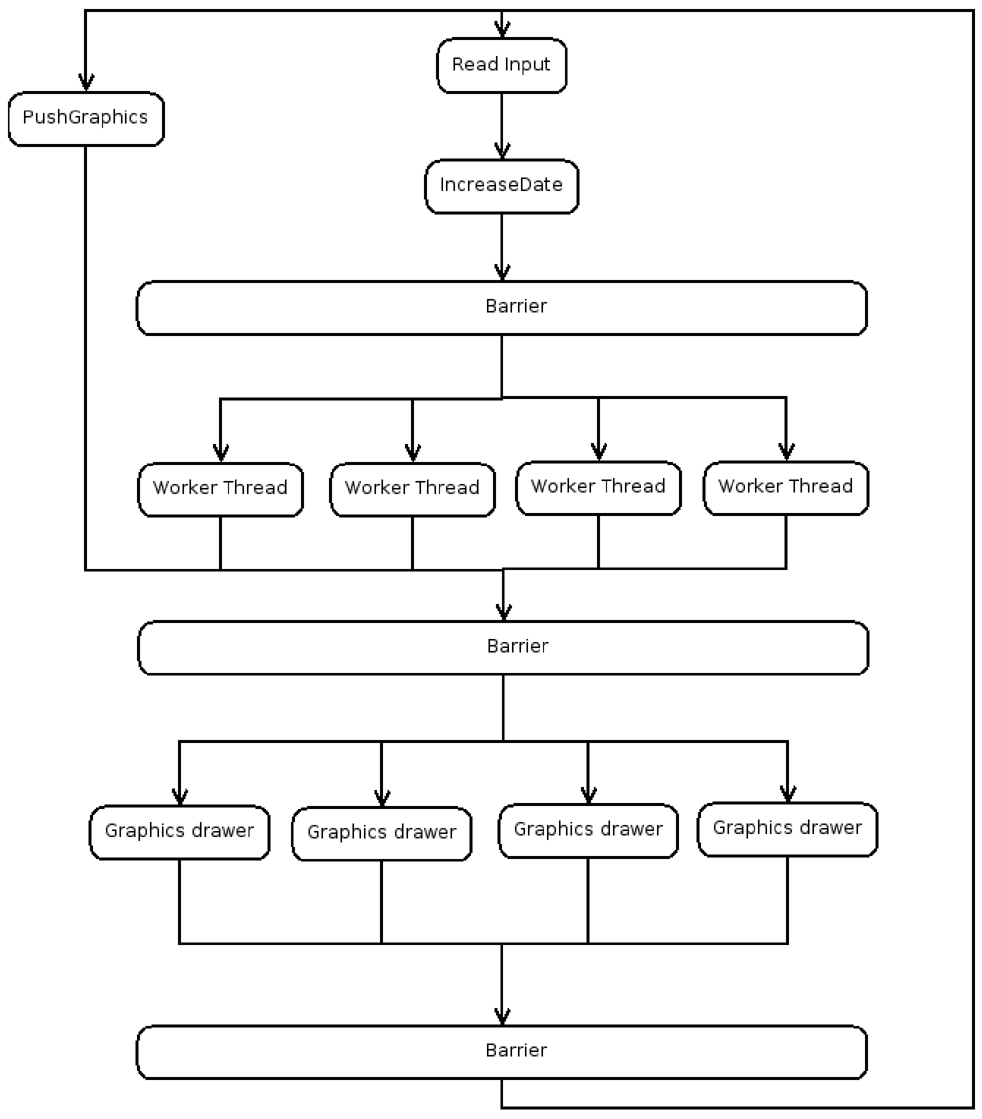

Figure 13: Redesign

8 Conclusion

In this report, we looked at issues related to parallelizing computer games. As a test case we use an open source game (OpenTTD). More CPUs are now being delivered with several cores. Games can greatly benefit from this. You now have more processing power to create better Artificial Intelligence, use physics to create more realistic games, and larger and more complex games. It is a must that games take advantage of this, because it will give you an edge compared to competitors. This study looked at how one can parallelize a game so that one can take advantage of the extra processing power without rewriting the game.

Because OpenTTD is a game that is not multithreaded, we had to look at the source code, and analyze it, including measuring time spent in various loops and functions. It has to be done that way, because only then you will know what is worth parallelizing and what is not.

We found that if the game should function both on a single core and dual core CPU, including network playing, the following of our multithreaded implementations work together. First sound updates, where all sounds are pushed into a queue, and one thread dequeues the sounds and plays them. The second is pushing of the pixel array to the graphics card, which takes longer than the game loop. With a thread doing this, the main thread which includes the game loop will actually wait for the thread to finish. This opens for a more heavy game logic, including a more complex AI and better pathfinding.

The best way to parallelize the whole game would, however, be to rewrite the entire game from the ground up. It would then be important to always think of multithreading, so that loops and functions could be run independently of each other. This is not how it is done today, as discussed in the previous section. If the code were to utilize more CPUs than what we have managed to do, the best approach would be to rewrite the loop UpdateWindows(), so that it can be run with more than one thread. This means that two or more threads update different parts of the pixel array.

The techniques used here can easily be adapted to other games.

References

Books

[1B] John L. Hennessy and David A. Patterson
Computer Architecture A Quantitative Approach, 3rd Edition
Morgan Kaufman, 2003

[2B] Peter Prinz & Tony Crawford
C in a nutshell - A desktop quick reference
O'Reilly Media, Inc., 2006

Web pages and web articles

[1W] OpenTTD
OpenTTD open source game
`http://openttd.org/`
Last visited 29.08.2006

[2W] Original TTD
Original Transport Tycoon Deluxe game
`http://www.tycoongames.net/ttdpages.html`
Last visited 29.08.2006

[3W] GNU General Public License
The GPL
`http://www.gnu.org/copyleft/gpl.html`
Last visited 29.08.2006

[4W] Intel Core 2 Duo
Description of Intel Core 2 Duo
`http://www.intel.com/products/processor/core2duo/`
Last visited 30.08.2006

[5W] Doom 3 engine
Description of the doom 3 engine
`http://en.wikipedia.org/w/index.php?title=Doom_3_engine&oldid=80052181`
Last visited 31.10.2006

[6W] Unreal 3 engine
Description of the unreal 3 engine (version 3.0)
`http://en.wikipedia.org/w/index.php?title=Unreal_Engine&oldid=84792687`
Last visited 31.10.2006

[7W] Anandtech article of multithreading games
Chapter of article is a dialog between Anandtech and an unreal 3 engine developer, Tim Sweeney
http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377&p=3
Last visited 01.09.2006

[8W] Galactic Civilization 2
Forum for Galactic Civilization 2
http://apolyton.net/forums/showthread.php?threadid=73514
Last visited 10.10.2006

[9W] The Quest for More Processing Power, Part Two: "Multicore and multi-threaded gaming"
Article at Anandtech.com
http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377&p=4
Last visited 10.10.2006

[10W] Multithreaded terrain smoothing
Article about multithreaded terrain smoothing
http://gamasutra.com/features/20060531/gruen_01.shtml
Last visited 21.09.2006

[11W] Open Source Game Development
Open Source Game Development description from Intel
http://www.intel.com/cd/ids/developer/asmo-na/eng/254761.htm
Last visited 01.09.2006

[12W] OpenMP
Open Multi Processing website
http://www.openmp.org/
Last visited 01.09.2006

[13W] Practical examples of multi-threading in games
A presentation by Leigh Davies from Intel
http://www.ati.com/developer/brighton/07%20Intel%20Practical%20Multithreading.pdf
Last visited 05.09.2006

[14W] Multithreaded game engine architectures
A web article by Ville Mönkönnen
http://www.gamasutra.com/features/20060906/monkkonen_01.shtml
Last visited 26.10.2006

[15W] SDL Simple Directmedia Layer
Cross-platform media library.
http://www.libsdl.org/.
Last visited 29.08.2006

[16W] Posix Threads
Posix threads explained
http://www-128.ibm.com/developerworks/linux/library/l-posix1.html
Last visited 20.11.2006

Definitions

The following are special abbreviations and terms used in the text.

Abbreviation	Definition
AI	An abbreviation for Artificial Intelligence
API	An abbreviation for Application programming interface
CPU	An abbreviation for Central Processing Unit
DIRTY	If a tile is dirty, it means that it must be scheduled for a process to update it and redraw it to the screen
FPS	An abbreviation for Frames Per Second Another abbreviation is First Person Shooter (type of games)
PIXEL ARRAY	An array in memory representing pixel information for all pixels on the screen
SDL	An abbreviation for Simple DirectMedia Layer [15W]
SPRITE	A two-dimensional image or animation that is integrated into a larger scene
TESSELLATION	A tessellation or tiling of the plane is a collection of plane figures that fills the plane with no overlaps and no gaps
TICK	A step in the game logic
TYCOON	A person who controls a large portion of a particular industry and whose wealth derives primarily from said control
VIEWPORT	The current window that is viewed on the screen

Appendix A - Source Code

This section contains three code files, which demonstrates the code that gave the most significant speedup. Our modifications are marked with // OUR CODE and // END OUR CODE.

thread.c

```
/* $Id: thread.c 5978 2006-08-20 13:48:04Z truelight $ */

#include "stdafx.h"
#include "thread.h"
#include <sys/time.h>
#include <stdlib.h>

#if defined(__AMIGA__) || defined(__MORPHOS__) || defined(NO_THREADS)
OTTDThread *OTTDCreateThread(OTTDThreadFunc function, void *arg)
{ return NULL; }
void *OTTDJoinThread(OTTDThread *t) { return NULL; }
void OTTDExitThread(void) { NOT_REACHED(); };

#elif defined(__OS2__)

#define INCL_DOS
#include <os2.h>
#include <process.h>

struct OTTDThread {
  TID thread;
  OTTDThreadFunc func;
  void* arg;
  void* ret;
};

static void Proxy(void* arg)
{
  OTTDThread* t = arg;
  t->ret = t->func(t->arg);
}

OTTDThread* OTTDCreateThread(OTTDThreadFunc function, void* arg)
{
  OTTDThread* t = malloc(sizeof(*t));

  if (t == NULL) return NULL;

  t->func = function;
  t->arg  = arg;
  t->thread = _beginthread(Proxy, NULL, 32768, t);
  if (t->thread != -1) {
    return t;
  } else {
    free(t);
    return NULL;
  }
}
```

```
void* OTTDJoinThread(OTTDThread* t)
{
  void* ret;

  if (t == NULL) return NULL;

  DosWaitThread(&t->thread, DCWW_WAIT);
  ret = t->ret;
  free(t);
  return ret;
}

void OTTDExitThread(void)
{
  _endthread();
}

#elif defined(UNIX)

#include <pthread.h>

struct OTTDThread {
  pthread_t thread;
};

// OUR CODE

// added structs to wrap pthread variables/structs
struct OTTDMutex {
  pthread_mutex_t mutex;
};

struct OTTDCondition {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
};

struct OTTDBarrier {
  pthread_barrier_t barrier;
};

OTTDThreads _threads;

bool _done_graphics = false;

// return a thread id representing a unique thread
uint OTTDThreadid(void)
{
  return pthread_self();
}

// END OUR CODE

OTTDThread* OTTDCreateThread(OTTDThreadFunc function, void* arg)
{
  OTTDThread* t = malloc(sizeof(*t));
  if (t == NULL) return NULL;

  if (pthread_create(&t->thread, NULL, function, arg) == 0) {
    return t;
  } else {
    free(t);
    return NULL;
  }
}

void* OTTDJoinThread(OTTDThread* t)
{
  void* ret;

  if (t == NULL) return NULL;

  pthread_join(t->thread, &ret);
  free(t);
  return ret;
}

// OUR CODE

OTTDBarrier* OTTDBarrierInit(int number)
{
  OTTDBarrier* barrier = malloc(sizeof(OTTDBarrier));
  pthread_barrier_init(&barrier->barrier,(void*)"",number);
  return barrier;
}

void OTTDBarrierWait(OTTDBarrier* barrier)
{
  pthread_barrier_wait(&barrier->barrier);
}

// END OUR CODE
void OTTDExitThread(void)
{
  pthread_exit(NULL);
}

// OUR CODE
OTTDMutex* OTTDMutexCreate(void)
{
  OTTDMutex* mu = malloc(sizeof(OTTDMutex));

  if (mu == NULL) return NULL;

  if (pthread_mutex_init(&(mu->mutex),NULL) == 0 ) {
    return mu;
  } else {
    free(mu);
    return NULL;
  }
}

int OTTDMutexLock(OTTDMutex* mu)
{
  return pthread_mutex_lock(&(mu->mutex));
}

int OTTDMutexUnlock(OTTDMutex* mu)
{
  return pthread_mutex_unlock(&(mu->mutex));
}

int OTTDMutexDestroy(OTTDMutex *mu)
{
  int tmp = pthread_mutex_destroy(&(mu->mutex));
  free(mu);
  return tmp;
}

OTTDCondition* OTTDConditionCreate(void)
{
  OTTDCondition* co = malloc(sizeof(OTTDCondition));

  if (co == NULL) return NULL;

  if (pthread_mutex_init(&(co->mutex),NULL) == 0
    && pthread_cond_init(&(co->cond),NULL) == 0) {
    return co;
  } else {
    free(co);
    return NULL;
  }
}

int OTTDConditionSignal(OTTDCondition* co)
{
  return pthread_cond_signal(&(co->cond));
}

int OTTDConditionWait(OTTDCondition* co)
{
  return pthread_cond_wait(&(co->cond), &(co->mutex));
}

int OTTDConditionTimedWait(OTTDCondition* co, int msec)
{
  struct timeval now;
  struct timespec timeout;
  struct timezone tz;
  tz.tz_minuteswest = 0;
  tz.tz_dsttime = 0;
  gettimeofday(&now,&tz);
  timeout.tv_sec = now.tv_sec + (int) msec/1000;
  timeout.tv_nsec = now.tv_usec * 1000 + msec%1000 * 1000000;
  return pthread_cond_timedwait(&(co->cond), &(co->mutex), &timeout);
}

int OTTDConditonDestroy(OTTDCondition *co)
{
  int tmp = pthread_cond_destroy(&(co->cond));
  free(co);
  return tmp;
```

```
}

// END OUR CODE
#elif defined(WIN32)

#include <windows.h>

struct OTTDThread {
  HANDLE thread;
  OTTDThreadFunc func;
  void* arg;
  void* ret;
};

static DWORD WINAPI Proxy(LPVOID arg)
{
  OTTDThread* t = arg;
  t->ret = t->func(t->arg);
  return 0;
}

OTTDThread* OTTDCreateThread(OTTDThreadFunc function, void* arg)
{
  OTTDThread* t = malloc(sizeof(*t));
  DWORD dwThreadId;

  if (t == NULL) return NULL;

  t->func = function;
  t->arg  = arg;
  t->thread = CreateThread(NULL, 0, Proxy, t, 0, &dwThreadId);

  if (t->thread != NULL) {
    return t;
  } else {
    free(t);
    return NULL;
  }
}

void* OTTDJoinThread(OTTDThread* t)
{
  void* ret;

  if (t == NULL) return NULL;

  WaitForSingleObject(t->thread, INFINITE);
  CloseHandle(t->thread);
  ret = t->ret;
  free(t);
  return ret;
}

void OTTDExitThread(void)
{
  ExitThread(0);
}
#endif
```

sound.c

```
/* $Id: sound.c 5609 2006-07-26 03:33:12Z belugas $ */

#include "stdafx.h"
#include "openttd.h"
#include "functions.h"
#include "map.h"
#include "mixer.h"
#include "sound.h"
#include "vehicle.h"
#include "window.h"
#include "viewport.h"
#include "fileio.h"
#include "queue.h"
#include "thread.h"

typedef struct FileEntry {
  uint32 file_offset;
  uint32 file_size;
  uint16 rate;
  uint8 bits_per_sample;
  uint8 channels;
} FileEntry;

typedef struct SoundInfo {
  uint sound;
  int panning;
  uint volume;
} SoundInfo;

static uint _file_count;
static FileEntry *_files;
// OUR CODE
static Queue *soundqueue;
static void* SoundPlayer(void* arg);

static OTTDThread *soundthread;
static OTTDCondition *soundcondition;
// END OUR CODE

#define SOUND_SLOT 63
// Number of levels of panning per side
#define PANNING_LEVELS 16
// OUR CODE
extern OTTDThreads _threads;
// END OUR CODE
static void OpenBankFile(const char *filename)
{
  FileEntry *fe;
  uint count;
  uint i;

  FioOpenFile(SOUND_SLOT, filename);
  count = FioReadDword() / 8;
  fe = calloc(count, sizeof(*fe));

  if (fe == NULL) {
    _file_count = 0;
    _files = NULL;
    return;
  }

  _file_count = count;
  _files = fe;

  FioSeekTo(0, SEEK_SET);

  for (i = 0; i != count; i++) {
    fe[i].file_offset = FioReadDword();
    fe[i].file_size = FioReadDword();
  }

  for (i = 0; i != count; i++, fe++) {
    char name[255];

    FioSeekTo(fe->file_offset, SEEK_SET);

    // Check for special case, see else case
    FioReadBlock(name, FioReadByte()); // Read the name of the sound
    if (strcmp(name, "Corrupt␣sound") != 0) {
      FioSeekTo(12, SEEK_CUR); // Skip past RIFF header

      // Read riff tags
      for (;;) {
        uint32 tag = FioReadDword();
        uint32 size = FioReadDword();

        if (tag == '␣tmf') {
          FioReadWord(); // wFormatTag
          fe->channels = FioReadWord(); // wChannels
          FioReadDword();    // samples per second
          fe->rate = 11025;
          // seems like all samples should be played at this rate.
          FioReadDword();    // avg bytes per second
          FioReadWord();     // alignment
          fe->bits_per_sample = FioReadByte(); // bits per sample
          FioSeekTo(size - (2 + 2 + 4 + 4 + 2 + 1), SEEK_CUR);
        } else if (tag == 'atad') {
          fe->file_size = size;
          fe->file_offset = FioGetPos() | (SOUND_SLOT << 24);
          break;
        } else {
          fe->file_size = 0;
          break;
        }
      }
    } else {
      /*
       * Special case for the jackhammer sound
       * (name in sample.cat is "Corrupt sound")
       * It's no RIFF file, but raw PCM data
       */
      fe->channels = 1;
      fe->rate = 11025;
      fe->bits_per_sample = 8;
      fe->file_offset = FioGetPos() | (SOUND_SLOT << 24);
    }
  }
}

static bool SetBankSource(MixerChannel *mc, uint bank)
{
  const FileEntry *fe;
  int8 *mem;
  uint i;

  if (bank >= _file_count) return false;
  fe = &_files[bank];

  if (fe->file_size == 0) return false;

  mem = malloc(fe->file_size);
  if (mem == NULL) return false;

  FioSeekToFile(fe->file_offset);
  FioReadBlock(mem, fe->file_size);

  for (i = 0; i != fe->file_size; i++)
    mem[i] += -128; // Convert unsigned sound data to signed

  assert(fe->bits_per_sample == 8 && fe->channels == 1
    && fe->file_size != 0 && fe->rate != 0);

  MxSetChannelRawSrc(mc, mem, fe->file_size, fe->rate, MX_AUTOFREE);

  return true;
}

bool SoundInitialize(const char *filename)
{
  // OUR CODE
  soundqueue = new_Fifo(100);
  soundcondition = OTTDConditionCreate();
  soundthread = OTTDCreateThread(&SoundPlayer,(void*)filename);
  // END OUR CODE
  return true;
}

// Low level sound player
static void PlaySound(uint sound, int panning, uint volume)
{
  MixerChannel *mc;
  uint left_vol, right_vol;

  if (volume == 0) return;
  mc = MxAllocateChannel();
  if (mc == NULL) return;
  if (!SetBankSource(mc, sound)) return;

  panning = clamp(panning, -PANNING_LEVELS, PANNING_LEVELS);
  left_vol = (volume * PANNING_LEVELS) - (volume * panning);
  right_vol = (volume * PANNING_LEVELS) + (volume * panning);
  MxSetChannelVolume(mc, left_vol * 128 / PANNING_LEVELS,
        right_vol * 128 / PANNING_LEVELS);
  MxActivateChannel(mc);
}
// OUR CODE
static void* SoundPlayer(void* arg){
  SoundInfo *si;
  si = NULL;
  _threads.sound = OTTDThreadid();
  OpenBankFile(arg);
  while (true) {
    si = soundqueue->pop(soundqueue);
    if(si == NULL){
      OTTDConditionTimedWait(soundcondition,300);
    }
    if (si != NULL) {
      PlaySound(si->sound,si->panning,si->volume);
      free(si);
      si = NULL;
    }
  }
}

static void StartSound(uint sound, int panning, uint volume)
{
  SoundInfo *si = malloc(sizeof(SoundInfo));
  si->sound = sound;
  si->panning = panning;
  si->volume = volume;
  soundqueue->push(soundqueue,si,1);
  OTTDConditionSignal(soundcondition);
}
// END OUR CODE

static const byte _vol_factor_by_zoom[] = {255, 190, 134};

static const byte _sound_base_vol[] = {
  128,  90, 128, 128, 128, 128, 128, 128,
  128,  90,  90, 128, 128, 128, 128, 128,
  128, 128, 128,  80, 128, 128, 128, 128,
  128, 128, 128, 128, 128, 128, 128, 128,
  128, 128,  90,  90,  90, 128,  90, 128,
  128,  90, 128, 128, 128,  90, 128, 128,
  128, 128, 128, 128,  90, 128, 128, 128,
  128,  90, 128, 128, 128, 128, 128, 128,
  128, 128,  90,  90,  90, 128, 128, 128,
   90,
};

static const byte _sound_idx[] = {
   2,  3,  4,  5,  6,  7,  8,  9,
  10, 11, 12, 13, 14, 15, 16, 17,
  18, 19, 20, 21, 22, 23, 24, 25,
  26, 27, 28, 29, 30, 31, 32, 33,
  34, 35, 36, 37, 38, 39, 40,  0,
   1, 41, 42, 43, 44, 45, 46, 47,
  48, 49, 50, 51, 52, 53, 54, 55,
  56, 57, 58, 59, 60, 61, 62, 63,
  64, 65, 66, 67, 68, 69, 70, 71,
  72,
};

static void SndPlayScreenCoordFx(SoundFx sound, int x, int y)
{
  const Window *w;

  if (msf.effect_vol == 0) return;

  for (w = _windows; w != _last_window; w++) {
    const ViewPort* vp = w->viewport;

    if (vp != NULL &&
        IS_INSIDE_1D(x, vp->virtual_left, vp->virtual_width) &&
        IS_INSIDE_1D(y, vp->virtual_top, vp->virtual_height)) {
      int left = (x - vp->virtual_left);

      StartSound(
        _sound_idx[sound],
        left / (vp->virtual_width / ((PANNING_LEVELS << 1) + 1))
          - PANNING_LEVELS,(_sound_base_vol[sound] *
          msf.effect_vol * _vol_factor_by_zoom[vp->zoom]) >> 15
      );
      return;
    }
  }

}

void SndPlayTileFx(SoundFx sound, TileIndex tile)
{
  /* emits sound from center of the tile */
  int x = TileX(tile) * TILE_SIZE + TILE_SIZE / 2;
  int y = TileY(tile) * TILE_SIZE + TILE_SIZE / 2;
  Point pt = RemapCoords(x, y, GetSlopeZ(x, y));
  SndPlayScreenCoordFx(sound, pt.x, pt.y);
}

void SndPlayVehicleFx(SoundFx sound, const Vehicle *v)
{
  SndPlayScreenCoordFx(sound,
    (v->left_coord + v->right_coord) / 2,
    (v->top_coord + v->bottom_coord) / 2
  );
}

void SndPlayFx(SoundFx sound)
{
  StartSound(
    _sound_idx[sound],
    0,
    (_sound_base_vol[sound] * msf.effect_vol) >> 7
  );
}
```

sdl_v.c

```
/* $Id: sdl_v.c 6380 2006-09-04 17:30:30Z rubidium $ */

#include "../stdafx.h"

#ifdef WITH_SDL

#include "../openttd.h"
#include "../debug.h"
#include "../functions.h"
#include "../gfx.h"
#include "../macros.h"
#include "../sdl.h"
#include "../window.h"
#include "../network.h"
#include "../variables.h"
#include "sdl_v.h"
#include <SDL.h>

#include "../thread.h"

// OUR CODE
OTTDBarrier *videobarrier;
static OTTDThread *videothread;
void* VideoThread(void * );
int thread_exit = 0;
// END OUR CODE

static SDL_Surface *_sdl_screen;
static bool _all_modes;

#define MAX_DIRTY_RECTS 100
static SDL_Rect _dirty_rects[MAX_DIRTY_RECTS];
static int _num_dirty_rects;

static void SdlVideoMakeDirty(int left, int top, int width, int height)
{
  //printf("SDLVIDEOMakeDirty\n");
  if (_num_dirty_rects < MAX_DIRTY_RECTS) {
    _dirty_rects[_num_dirty_rects].x = left;
    _dirty_rects[_num_dirty_rects].y = top;
    _dirty_rects[_num_dirty_rects].w = width;
    _dirty_rects[_num_dirty_rects].h = height;
  }
  _num_dirty_rects++;
}

static void UpdatePalette(uint start, uint count)
{
  SDL_Color pal[256];
  uint i;

  for (i = 0; i != count; i++) {
    pal[i].r = _cur_palette[start + i].r;
    pal[i].g = _cur_palette[start + i].g;
    pal[i].b = _cur_palette[start + i].b;
    pal[i].unused = 0;
  }

  SDL_CALL SDL_SetColors(_sdl_screen, pal, start, count);
}

static void InitPalette(void)
{
  UpdatePalette(0, 256);
}

static void CheckPaletteAnim(void)
{
  if (_pal_last_dirty != -1) {
    UpdatePalette(_pal_first_dirty, _pal_last_dirty - _pal_first_dirty + 1);
    _pal_last_dirty = -1;
  }
}

static void DrawSurfaceToScreen(void)
{
  int n = _num_dirty_rects;
  if (n != 0) {
    _num_dirty_rects = 0;
    if (n > MAX_DIRTY_RECTS)
      SDL_CALL SDL_UpdateRect(_sdl_screen, 0, 0, 0, 0);
    else
      SDL_CALL SDL_UpdateRects(_sdl_screen, n, _dirty_rects);
  }
}

static const uint16 default_resolutions[][2] = {
  { 640,  480},
  { 800,  600},
  {1024,  768},
  {1152,  864},
  {1280,  800},
  {1280,  960},
  {1280, 1024},
  {1400, 1050},
  {1600, 1200},
  {1680, 1050},
  {1920, 1200}
};

static void GetVideoModes(void)
{
  int i;
  SDL_Rect **modes;

  modes = SDL_CALL SDL_ListModes(NULL, SDL_SWSURFACE +
          (_fullscreen ? SDL_FULLSCREEN : 0));

  if (modes == NULL)
    error("sdl: no modes available");

  _all_modes = (modes == (void*)-1);

  if (_all_modes) {
      // all modes available, put some default ones here
      memcpy(_resolutions, default_resolutions, sizeof(default_resolutions));
      _num_resolutions = lengthof(default_resolutions);
    } else {
      int n = 0;
      for (i = 0; modes[i]; i++) {
        int w = modes[i]->w;
        int h = modes[i]->h;
        if (IS_INT_INSIDE(w, 640, MAX_SCREEN_WIDTH + 1) &&
            IS_INT_INSIDE(h, 480, MAX_SCREEN_HEIGHT + 1)) {
          int j;
          for (j = 0; j < n; j++) {
            if (_resolutions[j][0] == w && _resolutions[j][1] == h) break;
          }

          if (j == n) {
            _resolutions[j][0] = w;
            _resolutions[j][1] = h;
            if (++n == lengthof(_resolutions)) break;
          }
        }
      }
      _num_resolutions = n;
      SortResolutions(_num_resolutions);
    }
}

static void GetAvailableVideoMode(int *w, int *h)
{
  int i;
  int best;
  uint delta;

  // all modes available?
  if (_all_modes) return;

  // is the wanted mode among the available modes?
  for (i = 0; i != _num_resolutions; i++) {
    if (*w == _resolutions[i][0] && *h == _resolutions[i][1]) return;
  }

  // use the closest possible resolution
  best = 0;
  delta = abs((_resolutions[0][0] - *w) * (_resolutions[0][1] - *h));
  for (i = 1; i != _num_resolutions; ++i) {
    uint newdelta = abs((_resolutions[i][0] - *w) * (_resolutions[i][1] - *h));
    if (newdelta < delta) {
      best = i;
      delta = newdelta;
    }
  }
  *w = _resolutions[best][0];
  *h = _resolutions[best][1];
}

extern const char _openttd_revision[];

#ifndef ICON_DIR
```

```
#define ICON_DIR "media"
#endif

#ifdef WIN32
/* Let's redefine the LoadBMP macro with because we are dynamically
 * loading SDL and need to 'SDL_CALL' all functions */
#undef SDL_LoadBMP
#define SDL_LoadBMP(file) SDL_LoadBMP_RW(SDL_CALL SDL_RWFromFile(file, "rb"), 1)
#endif

static bool CreateMainSurface(int w, int h)
{
  SDL_Surface *newscreen, *icon;
  char caption[50];

  GetAvailableVideoMode(&w, &h);

  DEBUG(driver, 1) ("sdl:␣using␣mode␣%dx%d", w, h);

  /* Give the application an icon */
  icon = SDL_CALL SDL_LoadBMP(ICON_DIR PATHSEP "openttd.32.bmp");
  if (icon != NULL) {
    /* Get the colourkey, which will be magenta */
    uint32 rgbmap = SDL_CALL SDL_MapRGB(icon->format, 255, 0, 255);

    SDL_CALL SDL_SetColorKey(icon, SDL_SRCCOLORKEY, rgbmap);
    SDL_CALL SDL_WM_SetIcon(icon, NULL);
    SDL_CALL SDL_FreeSurface(icon);
  }

  // DO NOT CHANGE TO HWSURFACE, IT DOES NOT WORK
  newscreen = SDL_CALL SDL_SetVideoMode(w, h, 8, SDL_SWSURFACE | SDL_HWPALETTE |
          llscreen ? SDL_FULLSCREEN : SDL_RESIZABLE));
  if (newscreen == NULL)
    return false;

  _screen.width = newscreen->w;
  _screen.height = newscreen->h;
  _screen.pitch = newscreen->pitch;

  _sdl_screen = newscreen;
  InitPalette();

  snprintf(caption, sizeof(caption), "OpenTTD␣%s", _openttd_revision);
  SDL_CALL SDL_WM_SetCaption(caption, caption);
  SDL_CALL SDL_ShowCursor(0);

  GameSizeChanged();

  return true;
}

typedef struct VkMapping {
  uint16 vk_from;
  byte vk_count;
  byte map_to;
} VkMapping;
```

```
#define AS(x, z) {x, 0, z}
#define AM(x, y, z, w) {x, y - x, z}

static const VkMapping _vk_mapping[] = {
  // Pageup stuff + up/down
  AM(SDLK_PAGEUP, SDLK_PAGEDOWN, WKC_PAGEUP, WKC_PAGEDOWN),
  AS(SDLK_UP,      WKC_UP),
  AS(SDLK_DOWN,    WKC_DOWN),
  AS(SDLK_LEFT,    WKC_LEFT),
  AS(SDLK_RIGHT,   WKC_RIGHT),

  AS(SDLK_HOME,    WKC_HOME),
  AS(SDLK_END,     WKC_END),

  AS(SDLK_INSERT, WKC_INSERT),
  AS(SDLK_DELETE, WKC_DELETE),

  // Map letters & digits
  AM(SDLK_a, SDLK_z, 'A', 'Z'),
  AM(SDLK_0, SDLK_9, '0', '9'),

  AS(SDLK_ESCAPE,     WKC_ESC),
  AS(SDLK_PAUSE,      WKC_PAUSE),
  AS(SDLK_BACKSPACE, WKC_BACKSPACE),

  AS(SDLK_SPACE,      WKC_SPACE),
  AS(SDLK_RETURN,     WKC_RETURN),
  AS(SDLK_TAB,        WKC_TAB),

  // Function keys
  AM(SDLK_F1, SDLK_F12, WKC_F1, WKC_F12),

  // Numeric part.
  // What is the virtual keycode for numeric enter??
  AM(SDLK_KP0, SDLK_KP9, WKC_NUM_0, WKC_NUM_9),
  AS(SDLK_KP_DIVIDE,     WKC_NUM_DIV),
  AS(SDLK_KP_MULTIPLY, WKC_NUM_MUL),
  AS(SDLK_KP_MINUS,      WKC_NUM_MINUS),
  AS(SDLK_KP_PLUS,       WKC_NUM_PLUS),
  AS(SDLK_KP_ENTER,      WKC_NUM_ENTER),
  AS(SDLK_KP_PERIOD,     WKC_NUM_DECIMAL)
};

static uint32 ConvertSdlKeyIntoMy(SDL_keysym *sym)
{
  const VkMapping *map;
  uint key = 0;

  for (map = _vk_mapping; map != endof(_vk_mapping); ++map) {
    if ((uint)(sym->sym - map->vk_from) <= map->vk_count) {
      key = sym->sym - map->vk_from + map->map_to;
      break;
    }
  }

  // check scancode for BACKQUOTE key, because we want the key
  //  left of "1", not anything else (on non-US keyboards)
#if defined(WIN32) || defined(__OS2__)
```

```
  if (sym->scancode == 41) key |= WKC_BACKQUOTE;
#elif defined(__APPLE__)
  if (sym->scancode == 10) key |= WKC_BACKQUOTE;
#elif defined(__MORPHOS__)
  if (sym->scancode == 0)  key |= WKC_BACKQUOTE;
  // yes, that key is code '0' under MorphOS :)
#elif defined(__BEOS__)
  if (sym->scancode == 17)  key |= WKC_BACKQUOTE;
#elif defined(__SVR4) && defined(__sun)
  if (sym->scancode == 60) key |= WKC_BACKQUOTE;
  if (sym->scancode == 49) key |= WKC_BACKSPACE;
#elif defined(__sgi__)
  if (sym->scancode == 22) key |= WKC_BACKQUOTE;
#else
  if (sym->scancode == 41) key |= WKC_BACKQUOTE; // Linux console
  if (sym->scancode == 49) key |= WKC_BACKQUOTE;
#endif

  // META are the command keys on mac
  if (sym->mod & KMOD_META)  key |= WKC_META;
  if (sym->mod & KMOD_SHIFT) key |= WKC_SHIFT;
  if (sym->mod & KMOD_CTRL)  key |= WKC_CTRL;
  if (sym->mod & KMOD_ALT)   key |= WKC_ALT;
  // these two lines really help porting hotkey combos. Uncomment to use
#if 0
  printf("scancode␣character␣pressed␣%d\n", sym->scancode);
  printf("unicode␣character␣pressed␣%d\n", sym->unicode);
#endif
  return (key << 16) + sym->unicode;
}

static int PollEvent(void)
{
  SDL_Event ev;

  if (!SDL_CALL SDL_PollEvent(&ev)) return -2;

  switch (ev.type) {
    case SDL_MOUSEMOTION:
      if (_cursor.fix_at) {
        int dx = ev.motion.x - _cursor.pos.x;
        int dy = ev.motion.y - _cursor.pos.y;
        if (dx != 0 || dy != 0) {
          _cursor.delta.x += dx;
          _cursor.delta.y += dy;
          SDL_CALL SDL_WarpMouse(_cursor.pos.x, _cursor.pos.y);
        }
      } else {
        _cursor.delta.x = ev.motion.x - _cursor.pos.x;
        _cursor.delta.y = ev.motion.y - _cursor.pos.y;
        _cursor.pos.x = ev.motion.x;
        _cursor.pos.y = ev.motion.y;
        _cursor.dirty = true;
      }
      break;

    case SDL_MOUSEBUTTONDOWN:
      if (_rightclick_emulate && SDL_CALL SDL_GetModState() & KMOD_CTRL) {
        ev.button.button = SDL_BUTTON_RIGHT;
      }

      switch (ev.button.button) {
        case SDL_BUTTON_LEFT:
          _left_button_down = true;
          break;

        case SDL_BUTTON_RIGHT:
          _right_button_down = true;
          _right_button_clicked = true;
          break;

        case SDL_BUTTON_WHEELUP:     _cursor.wheel--; break;
        case SDL_BUTTON_WHEELDOWN:   _cursor.wheel++; break;

        default: break;
      }
      break;

    case SDL_MOUSEBUTTONUP:
      if (_rightclick_emulate) {
        _right_button_down = false;
        _left_button_down = false;
        _left_button_clicked = false;
      } else if (ev.button.button == SDL_BUTTON_LEFT) {
        _left_button_down = false;
        _left_button_clicked = false;
      } else if (ev.button.button == SDL_BUTTON_RIGHT) {
        _right_button_down = false;
      }
      break;

    case SDL_ACTIVEEVENT:
      if (!(ev.active.state & SDL_APPMOUSEFOCUS)) break;

      if (ev.active.gain) { // mouse entered the window, enable cursor
        _cursor.in_window = true;
      } else {
        UndrawMouseCursor(); // mouse left the window, undraw cursor
        _cursor.in_window = false;
      }
      break;

    case SDL_QUIT: HandleExitGameRequest(); break;

    case SDL_KEYDOWN: /* Toggle full-screen on ALT + ENTER/F */
      if ((ev.key.keysym.mod & (KMOD_ALT | KMOD_META)) &&
          (ev.key.keysym.sym == SDLK_RETURN || ev.key.keysym.sym == SDLK_f)) {
        ToggleFullScreen(!_fullscreen);
      } else {
        _pressed_key = ConvertSdlKeyIntoMy(&ev.key.keysym);
      }

      break;

    case SDL_VIDEORESIZE: {
      int w = clamp(ev.resize.w, 64, MAX_SCREEN_WIDTH);
      int h = clamp(ev.resize.h, 64, MAX_SCREEN_HEIGHT);
      ChangeResInGame(w, h);
      break;
    }
  }
  return -1;
}

static const char *SdlVideoStart(const char * const *parm)
{
  char buf[30];

  const char *s = SdlOpen(SDL_INIT_VIDEO);
  if (s != NULL) return s;

  SDL_CALL SDL_VideoDriverName(buf, 30);
  DEBUG(driver, 1) ("sdl:␣using␣driver␣'%s'", buf);

  GetVideoModes();
  CreateMainSurface(_cur_resolution[0], _cur_resolution[1]);
  MarkWholeScreenDirty();

  SDL_CALL SDL_EnableKeyRepeat(SDL_DEFAULT_REPEAT_DELAY,
          SDL_DEFAULT_REPEAT_INTERVAL);
  SDL_CALL SDL_EnableUNICODE(1);
  return NULL;
}

static void SdlVideoStop(void)
{
  SdlClose(SDL_INIT_VIDEO);
}

static void SdlVideoMainLoop(void)
{
  uint32 next_tick = SDL_CALL SDL_GetTicks() + 30;
  uint32 cur_ticks;
  uint32 pal_tick = 0;
  int i;
  uint32 mod;
  int numkeys;
  Uint8 *keys;
  int time1,time2;
  // OUR CODE
  //Thread initialization
  videobarrier = OTTDBarrierInit(2);
  videothread = OTTDCreateThread(VideoThread,NULL);
  // END OUR CODE
  for (;;) {
    InteractiveRandom(); // randomness

    while ((i = PollEvent()) == -1) {}
    if (_exit_game) return;

    mod = SDL_CALL SDL_GetModState();
    keys = SDL_CALL SDL_GetKeyState(&numkeys);
#if defined(_DEBUG)
    if (_shift_pressed)
```

```
#else
    if (keys[SDLK_TAB])
#endif
    {
      if (!_networking && _game_mode != GM_MENU) _fast_forward |= 2;
    } else if (_fast_forward & 2) {
      _fast_forward = 0;
    }

    cur_ticks = SDL_CALL SDL_GetTicks();
    if ((_fast_forward && !_pause) || cur_ticks > next_tick)
      next_tick = cur_ticks;

    if (cur_ticks == next_tick) {
      next_tick += 30;

      _ctrl_pressed  = !!(mod & KMOD_CTRL);
      _shift_pressed = !!(mod & KMOD_SHIFT);
#ifdef _DEBUG
      _dbg_screen_rect = !!(mod & KMOD_CAPS);
#endif

      // determine which directional keys are down
      _dirkeys =
        (keys[SDLK_LEFT]  ? 1 : 0) |
        (keys[SDLK_UP]    ? 2 : 0) |
        (keys[SDLK_RIGHT] ? 4 : 0) |
        (keys[SDLK_DOWN]  ? 8 : 0);

      GameLoop();
      // OUR CODE
      //printf("gameLoopBarrier\n");
      OTTDBarrierWait(videobarrier);
      // END OUR CODE
      _screen.dst_ptr = _sdl_screen->pixels;

      UpdateWindows();
      // OUR CODE
      //OTTDBarrierWait(videobarrier);
      //int time3 = getTimeInMicroseconds();
      //printf("GameLoop(): %d\nUpdateWindows(): %d\n",time2-time1,time3-time2);
      // END OUR CODE
      if (++pal_tick > 4) {
        CheckPaletteAnim();
        pal_tick = 1;
      }
      // OUR CODE
      //time1 = getTimeInMicroseconds();
      //DrawSurfaceToScreen();
      OTTDBarrierWait(videobarrier);
      //time2 = getTimeInMicroseconds();
      //printf("DrawSurfacetoScreen %d\n",time2-time1);
      //END OUR CODE
    } else {
      SDL_CALL SDL_Delay(1);
      OTTDBarrierWait(videobarrier);
      _screen.dst_ptr = _sdl_screen->pixels;
      DrawTextMessage();
      DrawMouseCursor();
      // OUR CODE
      //time1 = getTimeInMicroseconds();
      OTTDBarrierWait(videobarrier);
      //time2 = getTimeInMicroseconds();
      //printf("DrawSurfacetoScreen %d\n",time2-time1);
      //END OUR CODE
    }
  }
  thread_exit = 1;
}
// OUR CODE
void* VideoThread(void* per){
  OTTDBarrierWait(videobarrier);
  while(!thread_exit){
    OTTDBarrierWait(videobarrier);
    DrawSurfaceToScreen();
    OTTDBarrierWait(videobarrier);
  }

}
// END OUR CODE

static bool SdlVideoChangeRes(int w, int h)
{
  return CreateMainSurface(w, h);
}

static void SdlVideoFullScreen(bool full_screen)
{
  _fullscreen = full_screen;
  GetVideoModes(); // get the list of available video modes
  if (!_video_driver->change_resolution(_cur_resolution[0], _cur_resolution[1])) {
    // switching resolution failed, put back full_screen to original status
    _fullscreen ^= true;
  }
}

const HalVideoDriver _sdl_video_driver = {
  SdlVideoStart,
  SdlVideoStop,
  SdlVideoMakeDirty,
  SdlVideoMainLoop,
  SdlVideoChangeRes,
  SdlVideoFullScreen,
};

#endif
```