Incremental Computer

Program designed and developed by:
Max Linke

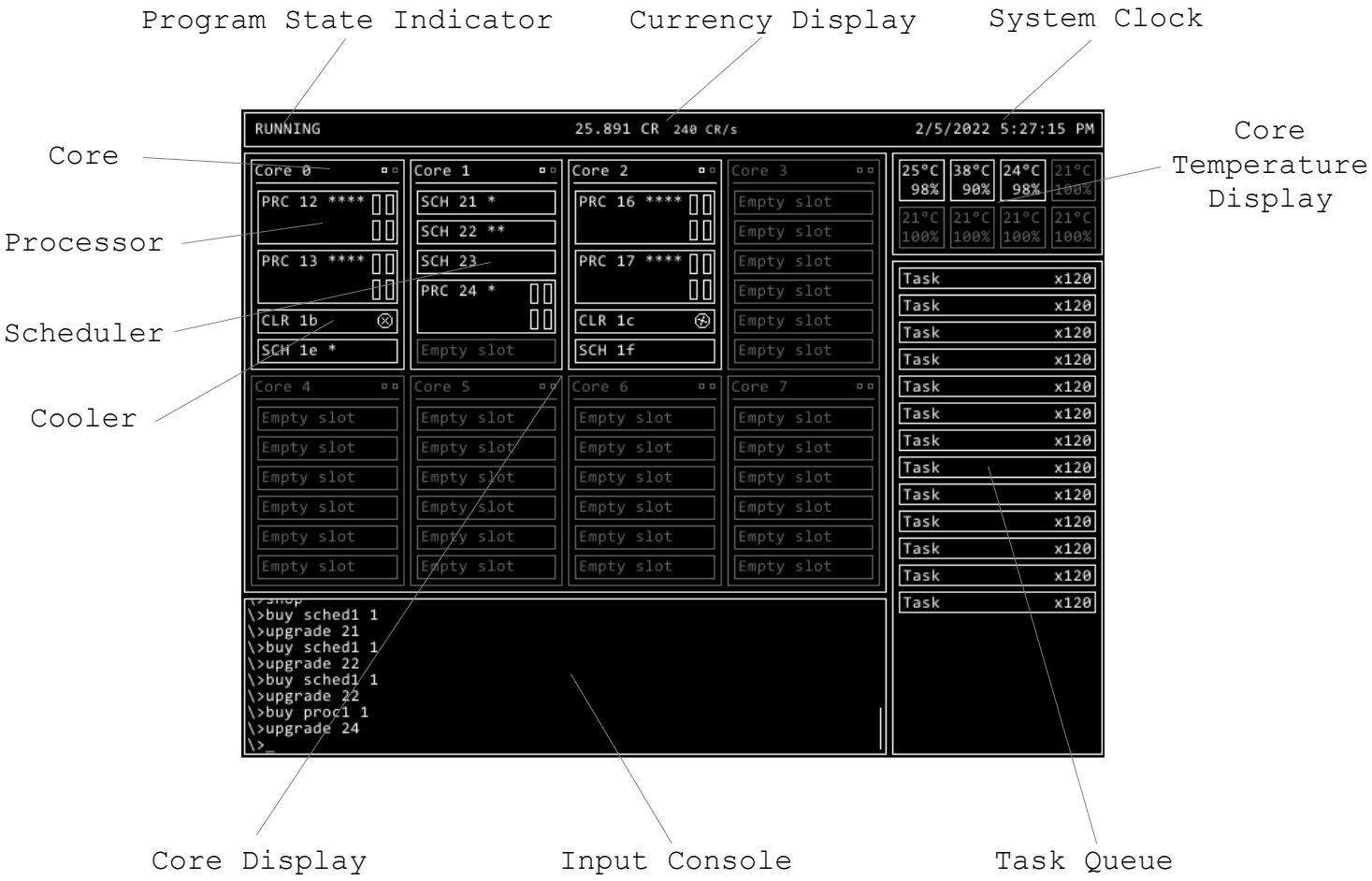
Technology used:
Unity Engine, UniRx Plugin [1]

Repository Link:
https://github.com/maxlinke/GT3-Incremental-Computer

Manual

As this game is trying to evoke the feeling of old DOS-era computer programs, all input is performed with the keyboard.

The interface and its elements are as follows:



Interactions with the program happen via commands typed into the console and confirmed by pressing Return. The list-command prints all available commands to the console. The help-command prints how a given command is to be used.

In addition to the main display, as shown above, there is also a shop, opened via the shop-command. While in the shop, all purchaseable items and upgrades are displayed. The pages are navigated by holding Ctrl and pressing either the left or right arrow key.

All purchases require a certain amount of CR, the program's internal currency. Currency is gained by schedulers creating tasks, which are then processed by processors.

To start the program, enter the run-command.

Progression

The initial program state consists of a single core with a single base-level, unupgraded processor and scheduler. The basic processor can process 4 tasks per cycle, however the basic scheduler only produces 1 task each time. This results in an effective gain of 1 CR/s at the start. The first steps, as soon as enough currency is available, should be purchasing more basic schedulers, as this is the cheapest way to increase CR/s.

As soon as the first core is filled with the basic processor and five basic schedulers however, the task queue begins to fill up faster than the processor can empty it. The next cheapest option to increase CR-gain is to upgrade the basic processor, so it takes six tasks each cycle. In the same way, the schedulers can be upgraded to produce more tasks each cycle. The final stage of the beginning could be a fully upgraded (****) processor, two fully upgraded (***) schedulers, one once-upgraded scheduler and two unupgraded schedulers. This produces 12 tasks each cycle, all of which are consumed by the processor yielding 12 CR/s.

With this gain, it becomes feasible to buy a second core to, at the very least, double one's gain by duplicating the pattern on the first core. However, it is possible to achieve triple the base gain by having one fully upgraded basic processor in each core, paired with 3 fully upgraded basic schedulers. This pattern can then be repeated one more time over the 2 free slots per core.

As the available CR increases, better components can be purchased, such as a level 1 processor, the base version of which can process 40 tasks per cycle. This effectively allows selling all three basic processors while gaining a free slot and the potential for even more gains.

This pattern of upgrading processors and schedulers in sync, buying cores when enough CR has been collected and thus increasing CR gain can be continued onward. However, the temperature generated by the processors at higher levels will cause their cores to throttle, reducing the cycle rate, necessitating the purchasing of coolers.

Theoretically, the game can be played until integer overflow occurs in the currency counter (and beyond), however with the way the program is currently set up, this will take far too long to achieve. A typical playthrough until the point where the CR gain becomes too low to buy components or upgrades in a reasonable amount of time is around 30 minutes.

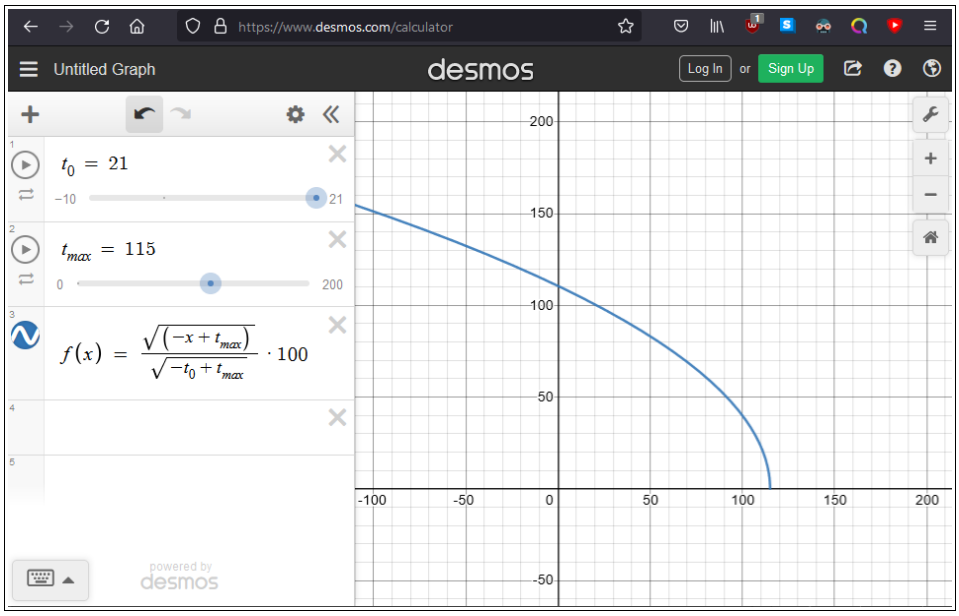
It is possible to get stuck by selling all CR-generating components and unlocking a core so that the remaining currency is insufficient for a basic processor and scheduler. This is on the user however, so there is no prevention in place.

Mathematics

As the shop deals in discrete purchases and the number of asterisks displayable on the components (indicating upgrades) is limited, so is the number of things the user can spend CR on.

The prices for core unlocks are truly exponential. The first one has a cost of 250 CR with a factor of 8 increase to the next in line. The prices for components and upgrades however have been hand-picked in order to allow somewhat smooth progression in the early game.

Formulas instead of hard numbers do appear in other places however. One of which is the temperature throttling present in the cores. A curve that doesn't punish the small temperature increases in the early game, while allowing for limited overclocking by actively cooling the processors under standard temperature, was desirable. The online calculator desmos [2] proved to be a brilliant tool for trying out various functions and parameters visually.



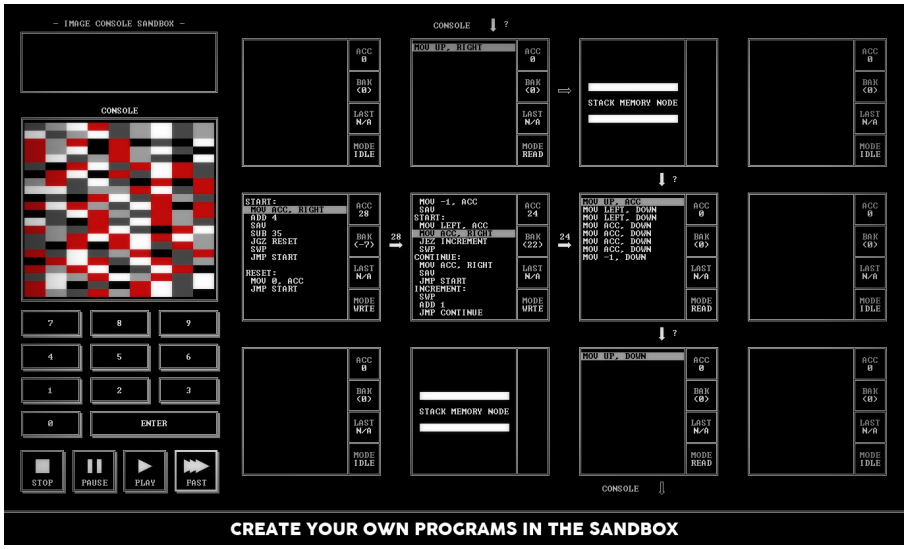
Regrettably, the discrete nature of the program (number of cores, number of components per core, currency gain per tick) coupled with the high degree of interlinking (number of schedulers to number of processors, thermal throttling) made it infeasible to use functions to calculate costs and gains easily. In the same way, it is not possible to generate an ideal progression graph.

Research

Due to unfamiliarity with the genre of incremental games and also a lack of interest, the gameplay inspirations were the best-known examples: Cookie Clicker and Universal Paperclips. With the simple idea that higher level purchases are simply better.

However with computers this isn't necessarily true. Early computers for example didn't need massive heat sinks and fans on their CPUs to keep from destroying themselves. Therefore higher level processors, capable of generating more CR have the drawback of requiring a cooler on the same core, if one wishes to keep the core from throttling to the point of near standstill.

Visually, there was one big inspiration: Zachtronic's TIS-100 [3].

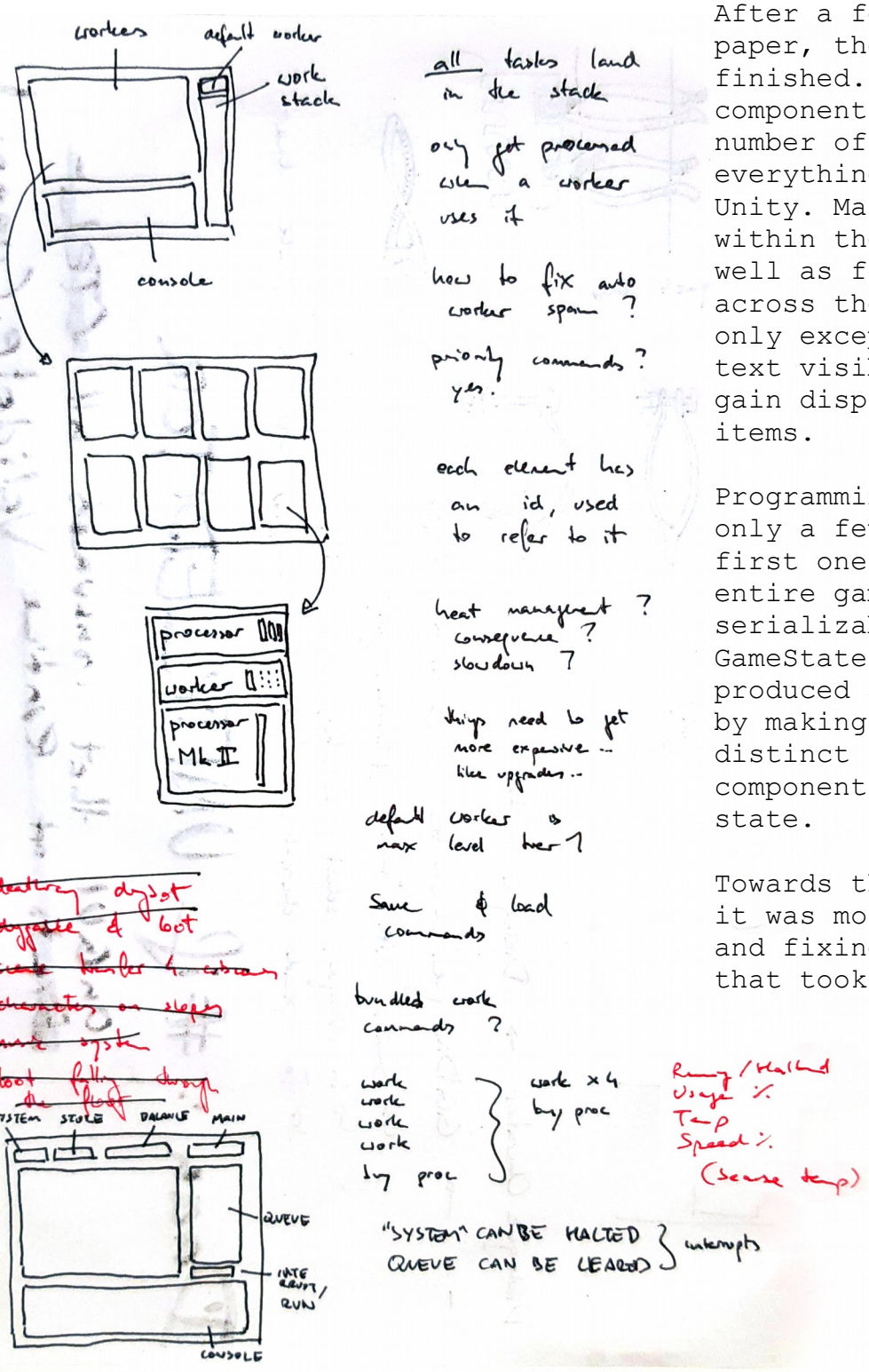


TIS-100 has the player program assembly-like code to perform a specific task. So the game is completely different, but the aesthetics were a stark influence. One immediately visible difference is however the use of color. TIS-100 uses red as a contrast to the otherwise black and white presentation. In this work, color has been omitted for more shades of grey, with grey usually representing something being unavailable, such as a not yet unlocked core or an expensive item in the shop.

Development

Being a one-man project, good resource management was paramount. The plan of spending as little time on art-assets and focusing on the programming resulted in the idea of a text-based incremental game. From this, the progression to a computer-terminal-based game to finally a program generating points for no discernable purpose came naturally.

To add more complexity than simply “buy more components, get more currency”, the distinction of processors and schedulers was devised, with the temperature mechanic and coolers adding another layer.



After a few design iterations on paper, the basic layout was finished. The precise number of components per core and maximum number of tasks depended on how everything fit together in Unity. Margins and padding within the various elements, as well as font size are unified across the entire program. The only exceptions being some small text visible in the currency gain display and info on shop items.

Programming went smoothly with only a few major refactors. The first one being transfer of the entire game state into a single serializable class named GameState. The second refactor produced a better MVC separation by making the component views distinct from the actual components present in the game state.

Towards the end of development, it was mostly adjusting numbers and fixing the occasional bug that took up time.

Context

The game is obviously an incremental game. The advice of using maths to drive costs and benefits of components wasn't taken due to complexity, the discrete nature of the interplay of numbers and the desire for "nice" numbers. Although this would have certainly allowed for the game to go on for longer than it currently does, possibly even making it an attainable goal to achieve integer overflow with the score as a sort of "win". At least the higher level purchases of components are both visually and effectively distinct from the lower levels to give a sense of reward when one is finally able to "level up".

Single components, such as the input console and the task queue were easy enough to program and test, constituting a kind of "prototyping", however the interlinked nature of game state, cores, components, where the numbers actually come from, and more only allowed the first "play tests" when the game was already half done. The interface was already fixed at this stage, so had it turned out that it is necessary to have 7 slots instead of 6 per core, that would have been bad. Luckily, since all the numbers are handpicked, it was possible to design around the six slot limit.

Various elements touched upon in the lecture (animation with lerps, procedural generation) were completely ignored in this project, because they didn't fit in with the theme. However, the more coding-oriented elements were regarded.

UniRx has been used, although sparingly. It is mostly the input console that provides subjects related to the various actions one can perform with it. Most of the reactive programming was done with simple C# events, for three reasons:

1. They don't require an import that causes confusion in VS Code, because both UniRx and the game have a class called "Scheduler".
2. They can be done with simple actions without parameters.
3. Most subscriptions don't filter, select, bundle or do anything on the input before executing the actual callback code, so it wasn't necessary.

The SOLID principles were partially used and partially ignored, depending on the situation. The MVC-separation of the core-components (Processor, Scheduler, Cooler) as well as the core itself is very much in favor of the Single Responsibility Principle, as is having a class for the game state. However sometimes it simply made sense to bundle things in one, such as the input console that both gets input from OnGUI-callbacks and then displays with Unity UI elements.

Finally, the code should be clean enough to be considered self documenting. No script is excessively long and methods are generally easy to understand by their signature.

Remarks

The refactor to keep the entire game state in a single serializable object was a good decision, as it allowed simplification of saving and loading the entire game on a whim. The MVC-refactor similarly made the code a lot more readable.

The shop with all its items being a single scriptable object, where prices and component-properties can be seen and tweaked easily beneficial when tweaking prices and performance towards the end of development.

It would have been nice to have a more fully-featured input console, with a moveable cursor, auto-complete and other nice-to-haves. However as it stands, it is functional and at least has a command memory allowing easy repeats for actions like repeatedly buying a certain type of component or upgrading the same component to max-level.

It was a huge moment of relief when the program was in a somewhat advanced state for the first time with multiple cores and different internal layouts therein. The asynchronous flashing of the various components, due to different purchasing times of cores and temperature-induced throttling made for a nice busy sight.

Other things that would have been nice although they provide no benefit to the game might be selectable color filters to emulate amber CRT tubes for example or a soundscape consisting of floppy drive noises and other retro-computing audio.

An attempt was made to have the various core components (Processor, Scheduler), self update via UniRx-calls, instead of getting OnUpdate-calls from their views, however this proved tricky and was abandoned in favor of the latter, actually functioning method.

References

[1] UniRx
https://github.com/neuecc/UniRx

[2] Desmos calculator
https://www.desmos.com/calculator

[3] TIS-100 Steam Page
https://store.steampowered.com/app/370360/TIS100/