Performance Effect of Flyweight in Soft Real-Time Applications

Erik S. V. Jansson
(*erija578@student.liu.se*)
Linköping University, Sweden

October 30, 2015

Abstract

Performance in soft real-time applications (e.g. video games and scientific simulations) is relevant since these must satisfy real-time constraints. The flyweight design pattern is claimed to reduce the memory consumption, a resource constraint.

The purpose of this paper is to analyze the effect of the flyweight pattern in real-time applications (primarily in video games) regarding performance. Measurements are made pertaining to the memory consumption and runtime timing properties.

Data is gathered by the means of a performance benchmark, several trials testing the relative difference between an implementation using the flyweight design pattern and another when not.

This data is then used to construct graphs that display the following properties: *reduced memory usage when the amount of shared data instantiated is significant* and *decreased runtime overhead in program sections taking advantage of the beneficial properties of the flyweight design pattern.*

Prerequisites

Knowledge of *software development* and *common terms* and *acronyms* used within. Comfortable in *object-oriented programming*, preferably *modern C++*. *Computer architecture* knowledge is beneficial in certain sections. Refer to literature such as *C++ Primer* [LLM12] by Lippman and *Computer Organization and Architecture* [Sta12] by Stallings.

Contents

1 Introduction

The difference in performance caused by applying the *flyweight design pattern* in many *soft real-time applications* may not be immediately obvious, but as will be seen, there is a relationship (in more ways than one). Before hypothesizing why, it is useful to describe *real-time systems* and the *flyweight* individually since they are the building blocks of the paper and require clear definitions and motivation.

1.1 Real-Time Systems

Software systems that are subject to *real-time* constraints are called *real-time systems* [Juv98]. These are systems that need to complete their task within a certain *deadline* in order to be useful. Real-time systems have become an essential component in our society, in fact, several parts of the modern world depend on these operating correctly.

Medicinal equipment (with accompanying software) is an excellent example of a *hard real-time system* that requires precise timing and system stability in order to guarantee human safety. Another such system is the stock market, where the loss isn't human, but monetary. Other systems where the consequences are not as severe are called *soft real-time systems*, where only the utility and quality of the system is degraded (not a failure).

Only *soft real-time applications* are going to be studied here. More specifically, the domain that is primarily going to be explored (and will thus be the focus) are *video games*. Games are soft real-time since a deviation from optimal operation only implies a degradation in system quality (usually in the form of fewer screen or game world updates).

1.2 The Flyweight Pattern

First officially coined in the article *Glyphs* [CL90], the *flyweight* design pattern was formally introduced in the famous *Design Patterns* [GHJV94] book by the authors commonly referred to as the *Gang of Four*. The book describes it succinctly: *"Use sharing to support large numbers of fine-grained objects efficiently" – flyweight description.*

Not surprisingly, similar techniques were already being used in practice long before being formally defined, one such was *ET++* [WGM88] written in 1988 by researchers exploring *GUIs* with *OOP*.

Flyweight objects can minimize the amount of redundant copies of a certain resource/object by reusing it. This is achieved by separating the *common* and *specific data* of an object. Common data is the subset of the object that remains static and can thus be shared between objects of the same type, only requiring a single instantiation. The specific data is used to differentiate between objects, usually by modifying parameters that change how the common data is interpreted and thus behaves.

Consider the following example: a virtual forest, every single tree has a *3D model* describing how it's going to be draw on the screen. This forest is quite boring, so all trees look very similar, therefore all could use the *same* 3D model. The only parameter that needs to be different is the *position*. By using a flyweight, one can separate the data for a tree in two parts: the common 3D model for all trees and the specific position for each one. This is an adaptation from *Game Programming Patterns* [Nys14], a book describing useful design patterns for games.

The *motivation* for choosing specifically the flyweight for analysis, instead of other patterns is because it relates closely to video games (the primary domain to be tested). This pattern is usually used unconsciously by game developers, since it seems logical that by reusing certain resources, *performance* and *memory benefits* may occur. By providing *empirical evidence*, this paper attempts to verify these claims and provide the resulting data.

1.3 Hypothesis and Question

Soft real-time applications need to satisfy real-world *timing* and *resource* constraints in order to be useful. Applications instantiating a substantial amount of similar objects (that have common data) can be divided into parts by using a flyweight, this will reduce the allocation of data by sharing that which is common. Therefore, by that reasoning, the application of the flyweight design pattern in the context of soft real-time applications should help better fulfill the resource constraint problem.

The purpose of this paper is to assert the validity of the above implication, while also gathering data to provide results regarding timing so a conclusion can be drawn whether the flyweight design pattern improves performance for soft real-time applications that instantiate similar objects. *Glyphs* [CL90] have also done similar experiments.

2 Method

In order to provide answers to the research questions posed in section 1.3, a *quantitative method* will be used, relying mostly on *empirical results* and a hands on (practical) approach. These data will be collected by the means of a *benchmark*, a program designed to assess *relative* performance by executing several trials measuring the desired properties.

2.1 Benchmarking Environment

Since the domain of computer games has specifically been selected for analysis (within soft real-time systems, see section 1.1), an appropriate set of trials needs to be constructed, each reflecting and testing relevant properties in such applications.

While many modern video games are presented from a *three-dimensional perspective*, there is still a strong and thriving market (especially for *independent* game developers) for *two-dimensional* games. The concept of a *sprite* is commonplace in such games, this is a two-dimensional image that represents some entity in the game world (possibly with accompanying animation, a *sprite sheet*).

Similar to the example given in section 1.2 of the virtual 3D forest, many of these sprites are going to instantiate redundant information that could instead be shared between instances. The data that is going to be shared in this case is the image (or more formally, *texture*) representing the object. Specific data to a sprite instance may include: position, rotation, velocity, angular velocity and object center.

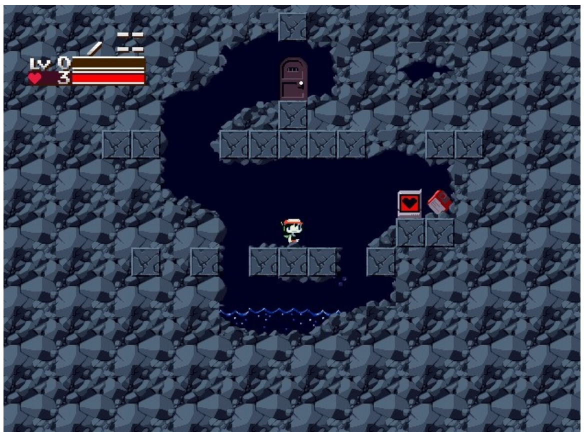

Figure 1: Cave Story+ developed by Nicalis.

For instance, look at figure 1. The components of this game world are divided into a grid of sprites, usually called *tiles* in such a context. As can be seen, several of these tiles use the exact same texture, the only difference between them is the position they are specified. By using something akin to the flyweight design pattern, the game engine does not need to instantiate redundant copies of the textures and instead reuses them for other tiles.

This is the behavior that will be targeted in the benchmark, by creating a substantial amount of sprites and measuring the difference in performance one can assess the effect of the flyweight design patten in video games. More specifically, the trial will be a virtual *asteroid field* that is bound to an area (the size of the computer screen in this case).

All these asteroids will look similar and hence use the *same texture*, each will also contain *internal information* about *position*, *rotation* and *velocity*. Each asteroid will have an initial state that is random, both position and velocity will be generated upon creation. Every cycle in the game loop will displace the asteroids by their respective velocities and be analyzed/handled for collision, finally, these are displayed on the screen with their texture (that might be shared via the flyweight design pattern). See algorithm 1 below, it describes roughly what the benchmarking environment will be doing.

Algorithm 1 Asteroid field benchmark trial.

```
Require: $count$, $texture$, $flyweight?$
  $asteroids \leftarrow \varnothing$
  populate($asteroids$, $count$)
  share?($texture$, $\forall a \in asteroids$, $flyweight?$)
  initialize($\forall a \in asteroids$, $rndstate$)
  repeat
    $exit \leftarrow$ input($user$)
    for ($\forall a \in asteroids$) do
      check-collision($a$, $screen$)
      handle-collision($a$, $a_v$)
      displace($a$, $a_v$)
      rotate($a$, $a_\theta$)
    end for
    display($\forall a \in asteroids$)
  until ($exit = yes$)
```

The relevant implementation details of the above algorithm are presented in section 2.3. Next, the performance properties of the trial are presented.

2.2 Properties for Analysis

Observing the structure of algorithm 1 reveals that there are *three* distinct stages the trial can be in: *initializing*, *updating* and *rendering*. All of these affect the applications *functional quality* [Cha13], more precisely, the *performance quality*. Based on these, the following metrics have been chosen:

- **Allocation time:** how long it takes (in *milliseconds*) for the trial to instantiate the list of sprites that represent the asteroids. Also, the texture itself needs to be allocated, by not sharing it with a flyweight the program needs to create one for each new asteroid.
- **Memory usage:** the amount of storage required to execute the trial, measured in *kilobytes*. Because most textures today are instantiated directly on the *graphics processing unit's video memory*, this will also be measured.

$$memory = count \times \text{sizeof}(asteroid) + \\ + \begin{cases} \text{sizeof}(texture), \ \textbf{if} \ flyweight \\ \textbf{else} \ count \times \text{sizeof}(texture) \end{cases}$$

- **Update time:** duration taken for all asteroids to transition to their next state, measured in *milliseconds*. The data used for the update is internal, no potentially shared data (texture in this case) via a flyweight is read or modified.
- **Render time:** how long it takes for all asteroids to be drawn on the screen. This will depend on how the common data is stored, that is, if flyweight is being used or not for sharing. This metric is of some importance for the paper, along with the next bullet point, will provide the information required to assess the effect of flyweight on the timing constraint.
- **Frame rate:** the amount of *frames* (renders of the entire scene) *per second*, which can also be expressed as the *frame frequency* in *hertz*. This metric is affected by both the update and render time in the course of one second. The most common measure of performance in games.

Data gathered from the benchmark will contain all of the above metrics, these are presented later in section 3. The conclusion presented depends heavily on these not being skewed performance metrics.

2.3 Implementation

Before presenting the results, a short rundown of the benchmark implementation is described below.

The chosen programming language for this task was *C++*, since it is commonly used in the industry for game development. Additionally, the simple media library *SDL2* was used to assist in rendering. The source code in its entirety can be found at: `gitlab.ida.liu.se/erija578/benchmark`.

First, the relevant structure and operations of the `Sprite` and `Sprite_factory` are presented. These are the equivalents of the `Flyweight` and `FlyweightFactory` constructs presented in the *Design Patterns* book [GHJV94, p. 198], but are different in some aspects (e.g. textures are stored):

```
class Sprite {
public:
    virtual ~Sprite() = default;
    void push(double, double);
    void move(double, double);
    void rotate(double);
    virtual void update();
    void render(SDL_Renderer*) const;
private:
    SDL_Rect source_{0, 0, 0, 0};
    SDL_Rect destination_{0, 0, 0, 0}
    SDL_Texture* texture_{nullptr};
    SDL_Point center_{0, 0};
    double angle_{0.0};
};
```

As with most of this section, several parts of the code have been left out for brevity. The size of the above structure is reported to be 104 bytes. Upon closer inspection, this should be smaller, the reason for this is because the compiler needs to *align* the data structure by applying *padding*, see [Dre07]. The only data that can be shared in the `Sprite` is the `SDL_Texture` as mentioned in section 2.1, all other fields need to be allocated with the `Sprite`. Instantiations are made by the `Sprite_factory`:

```
class Sprite_factory final {
public:
    Sprite create(SDL_Rect dest,
        const std::string& path,
        SDL_Renderer* renderer);
private:
    // Texture size: 1990 bytes.
    // Hash table: path -> resource.
    std::unordered_map<std::string,
        SDL_Texture*> textures_;
};
```

Following the reasoning presented in algorithm 1, the implementation is largely divided into three parts. There are several steps that have been added in comparison to the algorithm, most of these are concerned with gathering timing information and providing the results at the end of the benchmark.

Note, in the first three excerpts below that all start by executing `SDL_GetTicks()`, storing the time when each sequence of operations (*initializing*, *updating*, *rendering*) started and then, when finished, calculating the total time taken. All have the following format: $time = time_{end} - time_{start}$.

```
unsigned alloc_start{SDL_GetTicks()};
std::vector<Sprite> sprites;
Sprite_factory sprite_factory;
for (std::size_t i{0}; i < amount; ++i) {
    SDL_Rect dest{wdist(prng), hdist(prng),
        TEXTURE_WIDTH, TEXTURE_HEIGHT};
    if (!strcmp(bench, "flyweight")) {
        sprites.push_back(sprite_factory.
            create(dest, TEXTURE_PATH,
                renderer));
    } else if (!strcmp(bench, "nflyweight")) {
        SDL_Texture* texture{
            IMG_LoadTexture(renderer,
                "share/texture.png")};
        sprites.emplace_back(dest, texture); }
}

unsigned alloc_end{SDL_GetTicks()};
unsigned alloc_time{alloc_end -
    alloc_start};
```

When initializing the benchmarking environment, two important structures are created: `std::vector<Sprite>` and `Sprite_factory`. As described before, several `Sprite` objects are going to be created to represent individual asteroids, the texture for these is stored and shared in `Sprite_factory` if testing the flyweight. Otherwise, if the goal of the current trial is to asses the performance when *not* implementing the flyweight, the texture is re-instantiated for each new `Sprite`.

```
unsigned update_start{SDL_GetTicks()};
for (auto& sprite : sprites) {
    // Displace, rotate and collide.
    sprite.update(); }
unsigned update_end{SDL_GetTicks()};
unsigned update_time{update_end -
    update_start};
update_time_sum += update_time;
```

Transitions each sprite to the next state by update.

```
unsigned render_start{SDL_GetTicks()};
SDL_RenderClear(renderer);
for (const auto& sprite : sprites)
    sprite.render(renderer);
SDL_RenderPresent(renderer);
unsigned render_end{SDL_GetTicks()};
unsigned render_time{render_end
    - render_start};
render_time_sum += render_time;
```

Upon rendering, the benchmark first clears the *frame buffer* by issuing `SDL_RenderClear()`, the `sprites` list is then iterated through to write the textures (wherever they are) to the frame buffer. Then, the scene is displayed to the user by calling `SDL_RenderPresent()`, which is *double buffered.* More detailed information about rendering can be found in literature such as Ingemar Ragnemalm's *Polygons Feel No Pain* [Rag13].

```
if (current - previous >= 1000) {
    framerate_sum += frames;
    update_sum += update_time_sum/frames;
    render_sum += render_time_sum/frames; }
```

After every second has passed in the benchmark, the gathered data up until this point is stored in the three variables displayed above. As can be seen, for the update and render measurements, the *sum* of these is collected every frame and is then divided by the number of frames, resulting in the *average* update/render time per frame. For the frame rate itself, the probed frequency is gathered at `frames`.

$$alloc_{time} = alloc_{end} - alloc_{start} \tag{1}$$

$$update_{avg} = \frac{update_{sum}}{duration} \tag{2}$$

$$render_{avg} = \frac{render_{sum}}{duration} \tag{3}$$

$$framerate_{avg} = \frac{framerate_{sum}}{duration} \tag{4}$$

Finally, the results collected throughout the benchmark for `duration` seconds will be written to the *stdout*, which is normally a text terminal. This is done by applying equations 1, 2, 3 and 4 to the gathered data presented earlier in this section.

These results are presented in the coming pages.

3 Results

By applying the aforementioned method described in section 2, results regarding the metrics presented in section 2.2 have been collected. In order to easier reproduce the testing conditions, below follows the test setup (running on a *GNU/Linux* distribution) and the settings on which the benchmark was run.

3.1 Testing Hardware

- **Intel i7 860 CPU:** up to 3.46 GHz clock rate, 4 × 32 KiB L1 split cache (instruction, data), 4 × 128 KiB L2 unified cache and 8 MiB L3.
- **AMD HD 5870 GPU:** 850 MHz clock rate, 20 × 8 KiB L1 texture cache, 4 × 128 KiB L2 shared cache and 1024 MiB GDDR5 VRAM.

While this information might seem excessive, some conclusions in section 4 depend on these being correct, particularly on the size of the *data caches*.

3.2 Benchmark Settings

These specific results pertain to the version of the benchmark that executes each trial for 10 seconds (to remove noise, giving average performance), for both *flyweight* and *non-flyweight* versions of it. The first trial in the benchmark begins by testing 100 sprites. This amount is then subsequently incremented by 100 sprites after each trial, returning the average performance and writing them to a file.

In the given implementation in section 2.3, there is a benchmark driver script called `run.sh`, the arguments that need to be given are: *number of trials* that are to be run, *step count* after each trial, *initial amount* of sprites and finally *probe time*. The results presented here were gathered with the following command: `./run.sh 100 100 100 10`.

The results are then output to the files: `flyweight.res` and `nflyweight.res` in the same directory as the script driver. Both of these contain the final results of each metric, divided in columns and for each trial, separated by rows. E.g:

```
Count    Alloc       Frames    Update    Render
  100      1ms    325.00 Hz    0.01ms    3.07ms
  200      1ms    249.10 Hz    0.02ms    3.99ms
  300      1ms    225.40 Hz    0.04ms    4.40ms
```

These are used to generate the upcoming graphs.

3.3 Allocation Time

Since the relative difference between the two datasets below increases by several magnitudes as the amount of sprites increases, the y-axis has been chosen to display data *logarithmically* (in base 10).

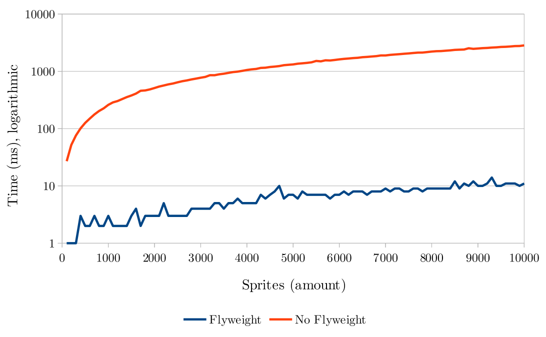

Figure 2: Graph for allocation time.

While the data does contain a lot of noise (these are not averaged), the relevant result provided here is that *the flyweight design pattern improves allocation time* given that enough data can be shared. This effect scales relative to the difference in allocation size between the specific and common data.

3.4 Memory Usage

Similar to the datasets presented in the previous section, these are also shown in logarithmic format.

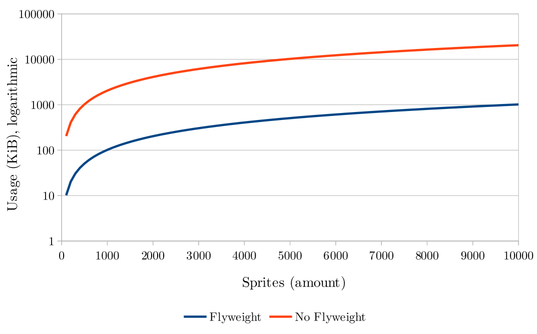

Figure 3: Graph for memory usage.

This is actually a sum of two storage types: standard memory (RAM) and video memory (VRAM). These results can be interpreted similarly as in the previous section, *the flyweight design pattern reduces memory usage* if there is data to be shared.

3.5 Update Time

As described in the end of section 2.1, the `update` procedure for each sprite does not take advantage of the sharing property of the flyweight, this since it only operates on its own specific data (no texture).

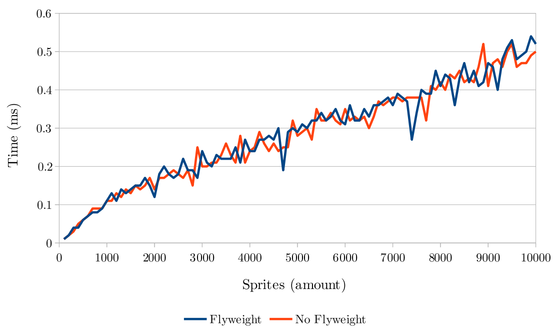

Figure 4: Graph for update time.

This is reflected by the datasets above, both flyweight and no flyweight results seem equivalent. Hence, *the flyweight design pattern does not affect update time in any meaningful way* given that `update` itself does not operate on the shared data.

3.6 Render Time

These datasets are the inverse of the above section, here the `render` procedure reads from the texture and also some of the specific data (position, angle).

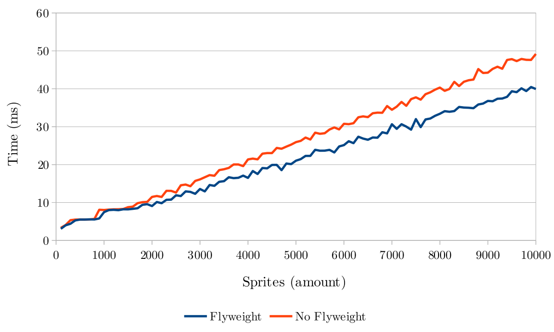

Figure 5: Graph for render time.

Because of these properties, *the average time taken to render a frame is steadily lower on the flyweight implementation.* This is presumed to hold only if the `render` method uses shared resources.

3.7 Frame Rate

This is the general metric for performance in video games, as described by the *Integrated CPU-GPU Power Management for 3D Mobile Games* [PJPM14] article. It combines all runtime overhead for updating and rendering, measured by the number of frames displayed in one second.

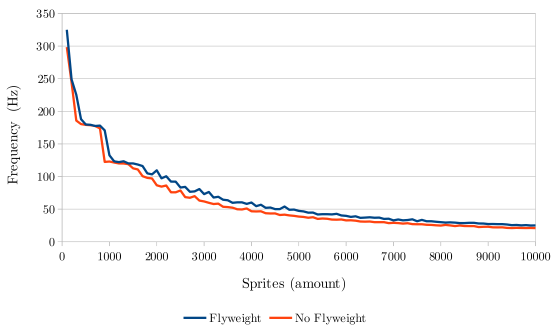

Figure 6: Graph for frame rate.

Given the above datasets, *the flyweight design pattern consistently provides higher frame rates* in relation to the non-flyweight version. Since frame rates are affected by both update time and render time, better render time implies better frame rate.

4 Discussion

In this section are reflections regarding section 3, what could be improved and conclusions (besides those presented in section 5) that are derived by the author but don't have sufficient quantitative proof to be stated as true in the final conclusion.

First, the data gathered here is mostly raw besides the averaging done on-line in the benchmark. While the results and conclusions (from them) are accurate, a *statistical analysis* could also have been done, giving more precise results and broader conclusions (e.g. providing performance increase rate).

Second, an interesting behavior in figure 6 is present somewhere between 800 and 1000 sprites, a considerable decline in frame frequency. The reason for this (as reported by *valgrind*) is that the CPU's L2 cache is full, and must resort to main memory (or the L3 8 MiB cache). If true, *the flyweight improves temporal cache locality*, this since the same data is queried, and can reside in the cache longer.

5 Conclusion

The research question posed in section 1 was: *how does the flyweight design pattern affect the performance of soft real-time applications, measured by memory usage and timing constraints?*

By applying the method presented in section 2, data is generated that is then displayed in section 3 in the form of graphs. These graphs provide performance comparisons between two implementations, one where flyweight is used and one were it isn't. If the aforementioned process is correct, the following answers to the question can be provided.

5.1 Memory Effects

By sharing common data, *application of the flyweight design pattern reduces memory usage.* This has only proven to be beneficial if the difference of the specific and common data is significant and a great amount of these have been instantiated. Similar assertions have been made by *Glyphs* [CL90]. The *Design Patterns* book [GHJV94] also provides claims of reduced memory usage in this case.

5.2 Timing Effects

Derived from the graphs in section 3, *the flyweight design pattern reduces runtime overhead* in sections of the program that utilize the shared/common portion of the object. More specifically, in the context of *video games*, this implied *lower allocation time*, *reduced average rendering time* and *increased average frame rate* when using a shared texture.

Draft Changes

Many of these improvements were in response to feedback given during the seminars. The author would particularly like to thank the project group, examinator and course staff for their help. The modifications since the initial draft are listed below:

- Specified what is analyzed (via method).
- Measurements are now specified (in method).
- Context for flyweight specified (in method).
- Most terms are implicit (via prerequisites).
- Completed remaining sections.

References

[Cha13] David Chappell. *The Three Aspects of Software Quality: Functional, Structural, Process.* Microsoft Corporation, 2013.

[CL90] Paul R. Calder and Mark A. Linton. Glyphs: Flyweight Objects for User Interfaces. *ACM User Interface Software Technologies Conference*, pages 92–101, 1990.

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory.* 2007.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Juv98] Kanaka Juvva. *Real-Time Systems.* Carnegie Mellon University, 1998.

[LLM12] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer.* Addison-Wesley Professional, 2012.

[Nys14] Robert Nystrom. *Game Programming Patterns.* Genever Benning, 2014.

[PJPM14] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated CPU-GPU Power Management for 3D Mobile Games. *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[Rag13] Ingemar Ragnemalm. *Polygons Feel No Pain.* Linköping University, 2013.

[Sta12] William Stallings. *Computer Organization and Architecture.* Pearson, 2012.

[WGM88] Andre Weinand, Erich Gamma, and Rudolf Marty. ET++ – An Object-Oriented Application Framework for C++. *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 46–57, 1988.