Game Design Patterns for CPU Performance Gain in Games

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Xi Wang
Matrikelnummer 1226083

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Bernhard Steiner

Wien, 23. August 2016

Xi Wang	Michael Wimmer

Game Design Patterns for Performance Gain in Games

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Xi Wang
Registration Number 1226083

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. Bernhard Steiner

Vienna, 23rd August, 2016

Xi Wang	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Xi Wang
Paminagasse 104, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. August 2016

Xi Wang

Kurzfassung

Diese Bachelorarbeit beschäftigt sich mit Design Patterns, welche in Computerspielen angewendet werden können, um die Performance zu verbessern. Dabei wurde ein ineffizientes Beispielprogramm in C++ implementiert, bei welchem vorallem die CPU zum Flaschenhals wird. Anschließend wurden verschiedene Design Patterns in das Programm eingebaut und der Performancegewinn in verschiedenen Kombinationen ausgewertet. Im Konkreten geht es um die Pattern: Flyweight, Object Pool, Component, Data Locality und Spatial Partitioning.

Abstract

This thesis looks into some Design Patterns that can be applied in game programming in order to improve performance. To measure the performance gain an inefficient sample program was implemented in C++, causing a CPU bottleneck. Afterwards patterns are applied to the program and the performance gain is measured for different combinations of patterns. In particular, the following patterns are investigated: Flyweight, Object Pool, Component, Data Locality and Spatial Partitioning.

Contents

CHAPTER 1

Introduction

Over the past few decades computers have improved tremendously. CPU computation time has been doubling every two year, though in recent years the percentage of improvement seems to decrease. While for many software development sectors this improvement has solved the problem of performance, in the branch of game development, however, performance is still an issue. Often GPU bottlenecks are the cause of performance problems, but the waste of CPU computation time due to poor software design is not to be underestimated, especially with computer games becoming more and more extensive and complex.

This thesis looks into a few design patterns that are used or can be used in game programming and measures their performance gain when using them. In particular, the Flyweight Pattern, Component Pattern, Data Locality, Spatial Partitioning and Object Pool Pattern are analyzed. The Flyweight Pattern is a structural pattern which improves performance by sharing common data between objects. The Component Pattern is a decoupling pattern, making the code more flexible. However this pattern can prevent deep inheritance hierarchy and therefore also might improve performance. Data Locality uses knowledge about the memory architecture for optimization while object pool helps to avoid unnecessary expensive memory allocations. Spatial partitioning splits the game world into smaller pieces in order to speed up calculations.

This thesis is structured as follows: First a description of the patterns and a more detailed explanation on how those patterns can improve performance is provided in Chapter 2. Chapter 3 describes the base implementation, which is a poorly designed game simulation, and how each pattern was included in order to improve the performance. Afterwards, for each tested combination, time was measured. Detailed test methodology are explained in Chapter 4. Results are discussed in Chapter 5.

CHAPTER 2

Design Patterns

2.1 Flyweight

The Flyweight Pattern is a structural pattern from the Gang of Fours [Joh94]. This pattern is very useful when an application needs a lot of objects that have common information between them. The typical OOP object has states. These states can be categorized into "intrinsic data" and "extrinsic data". Extrinsic data is information that depend on the context, for example, the position of some objects in the game world or individual health of a monster. Intrinsic data is data that is context independent, which would be the mesh of an object that can be drawn hundreds of times or a breed of monster that has certain characteristics. Therefore intrinsic data can be shared between many instances and can be encapsulated into a new object. This object can be shared and is referred to as the flyweight.

The Flyweight pattern helps to avoid unnecessary instantiation of shared data which saves instantiation time. Furthermore it saves memory since only one instance is loaded instead of hundreds, which might improve speed as well. The downside, however, is that each flyweight is a reference or pointer in the flyweight context, introducing an extra pointer indirection and possible cache miss (see Data Locality) and therefore again hurts the performance.

2.2 Object Pool

Dynamic memory allocation and deallocation in C++ can be a very slow operation. Firstly, the heap allocator can allocate memory of arbitrary size. However this requires a lot of extra management. Secondly, an (de-)allocation can force a context-switch from user-mode to kernel-mode and back which can be very expensive. [Gre14]

Object Pooling is one possibility to avoid expensive allocations. As the name suggests, the Object Pool pattern keeps a pool of objects. Instead of instantiating a new object each time when needed and freeing it afterwards the application "borrows" the object from the object pool. When the object is no longer needed, the application gives it back to the pool and the object can be reused. Though this pattern needs extra management of which objects are in use and is limited to a certain number of objects, it saves instantiation time, as well as memory allocation and freeing. Especially when many similar objects are created and released again, an object pool can increase performance.

Furthermore, using an Object Pool can prevent memory fragmentation. Memory fragmentation means that the available, non-allocated memory is split into smaller pieces instead of being one contiguous block. The resulting problem is that, when allocating memory that is bigger than the biggest contiguous memory available, the allocation will fail even if there is enough memory in total. The Object Pool allocates a contiguous memory, instead of allocating small memory blocks all over the place, and reuses the allocated memory, and prevents fragmentation of allocated and non-allocated memory.

2.3 Data Locality

While CPUs became faster and faster over the years, memory access stayed slow. In order to avoid long waits CPU caches were introduced. The typical memory hierarchy consists of registers, L1 cache, L2 cache, possible further levels of caches and main memory, ordered from fasted access time to slowest access time. A main memory access can take more than five hundreds of CPU cycles and while the data is fetched, the CPU wastes hundreds of cycles to wait for data, which is referred to as a CPU stall. In contrast a L1 cache access can take only around five to eight cycles [Llo11].

The underlying memory management works as followed: When main memory is accessed for a certain data, then bytes adjacent to this data are loaded too to fill a cache line. If for the next processing the CPU needs data and this data was already loaded into the cache, the CPU would not have to wait for data from the main memory. This is also called a "cache hit". If the data was not loaded in the cache then it is called "cache miss" and the main memory has to be accessed, which causes a CPU to wait more than five hundreds of cycles instead of five cycles.

The Data Locality Pattern uses this knowledge to improve performance. If the needed data is stored in the cache then the CPU does not have to wait for data, which results in a better performance. In order to make this cache loading behaviour to an advantage, data is stored in a contiguous memory block, putting as much data, that will be processed in the near future, into adjacent memory space to avoid cache misses. Cache misses were also one of the main reasons why today's game development is shifting from an object oriented design to a data oriented design [Llo11].

2.4 Spatial Partitioning

Spatial partitioning is often used to reduce the number of intersection or collision tests between objects and can improve performance drastically if there is a large number of objects. This pattern uses data structures to partition the world space. Objects in the game world are stored into a spatial data structure ordered by their position and therefore are separated from objects that are far away from each other. This helps to keep operations on objects locally instead of going through all objects.

The most simple spatial partitioning data structure is a grid. Other structures include Quadtree, Octree, Binary Space Partitioning Tree, Bounding Volume Hierarchy, k-d Tree and Spatial Hashing. Each structure has its advantages and disadvantages in a certain context, but all structures help to reduce the number of operations on objects. However using spatial partitioning means keeping movable objects in order.

2.5 Component

The component pattern is a decoupling pattern and often used in game engines nowadays (for example in Unity)[Nys14]. It is a pattern that helps to make the code more flexible. Instead of having one object that contains all behaviors and tasks, an object is a composition of many components, each having their own behavior and state. For example, an object can be rendered, which needs a mesh. It further can contain AI, which needs a game context. If the object represents a monster, it might contain information on how the monster moves, attacks and how it defends itself. Among different monsters there are many combinations of those behaviors. If all those behaviors were to be packed into one class it will become hard to manage. Furthermore, code for each behavior can not easily be reused with inheritance.

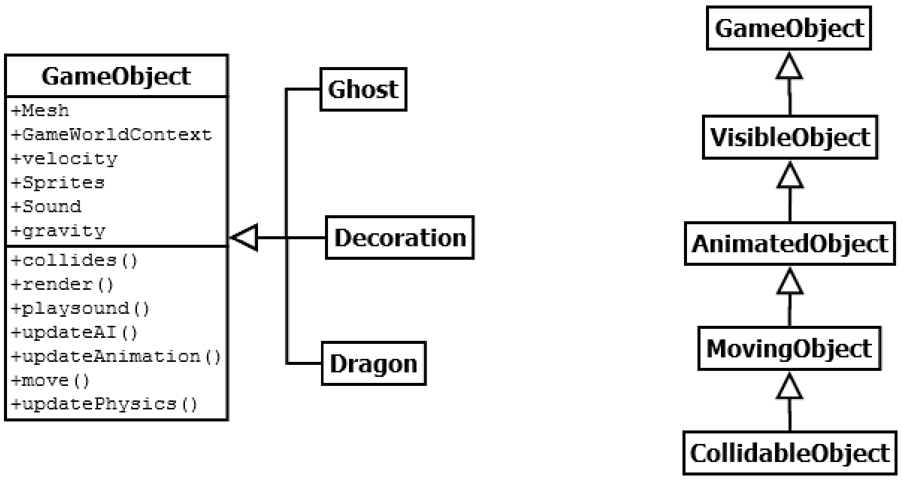

Figure 2.1: Possible class hierarchy designs.

Figure 2.1 shows two different design approaches without composition. The left one puts everything into the `GameObject` class, making it very big and probably unmanagable over time. Additionally deriving classes do not need all the inherited logic. A `Ghost` is not affected by physics and does not collide. Likewise `Decoration` does not need an AI. The design in Figure 2.1 (right) splits the huge class into smaller ones and uses inheritance for code reuse. A `Decoration` might inherit from `VisibleObject` and a `Ghost` from `MovingObject`. But what if a `Decoration` should be collidable but not movable? As one can see this design is limited as well.

A better approach would be to use composition as in Figure 2.2. With composition the object class would have a reference to its components. The `Dragon` would have a reference to a render-component, AI-component and different components for behavior. The `Ghost` would not need to have a `CollisionDetection` and `PhysicController`, while `Decoration` does not have an `AIComponent`. A `GameObject` can have any component it needs. At the same time code is reused.

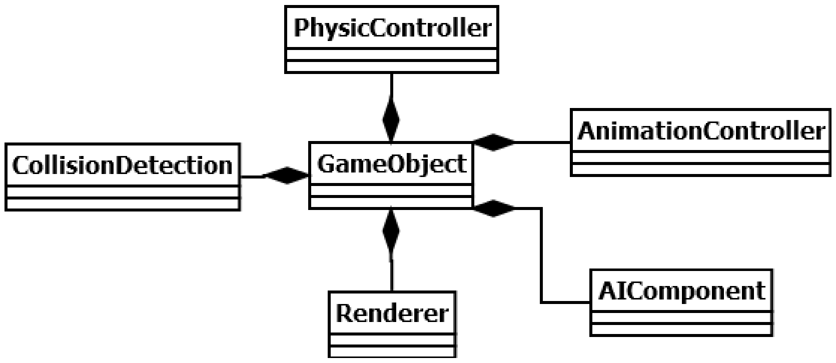

Figure 2.2: Possible component based design.

The component pattern is not known for performance optimization but rather for its flexibility for creating new objects with different behaviors. However, when inheritance is heavily used, with a deep inheritance tree and many virtual functions, a component based approach might also yield better performance. The reason for that is the extra overhead of V-Table lookups for each virtual function when using inheritance. With a component based approach only a reference or pointer is stored.

CHAPTER 3

Implementation

3.1 Libraries

The base 3D game simulation application was implemented in C++11 using OpenGL 3.3 [Ope16] as graphics API. Since OpenGL 3.3 was used GLEW [GLE16] had to be included as well. GLEW is the OpenGL Extension Wrangler and enables core functionalities as well as other extensions of OpenGL.

Furthermore SDL2 [SDL16a] was used. SDL2 is a cross platform library that enables easier access to OS dependent functionalities like window creation and keyboard and mouse input.

As the game scene uses complex 3D meshes a model loader was required. A well known model importer is the assimp library [Ass16]. Assimp can load models in various formats and stores the information into its own data structure. Using assimp often means parsing assimp's data structure into custom data structures, but this is still less work than writing a simple model loader.

A model contains textures. SDLimage [SDL16b] was used for texture loading.

For an easier use of mathematical computations the GLM library [GLM16] was also included. GLM provides vectors and matrices as data structures and many matrix and vector operations.

3.2 General description

In the simulation monsters are moving towards the center of the game world. Figure 3.1 shows a screenshot of the simulation. On their way there are towers shooting arrows that kill those monsters. Each time a monster reaches its destination or is killed by a tower the monster disappears and a new monster appears. Arrows disappear when they hit a target or the traveled distance is greater than a treshold. Collision is only applied between monsters and between monster and arrow.

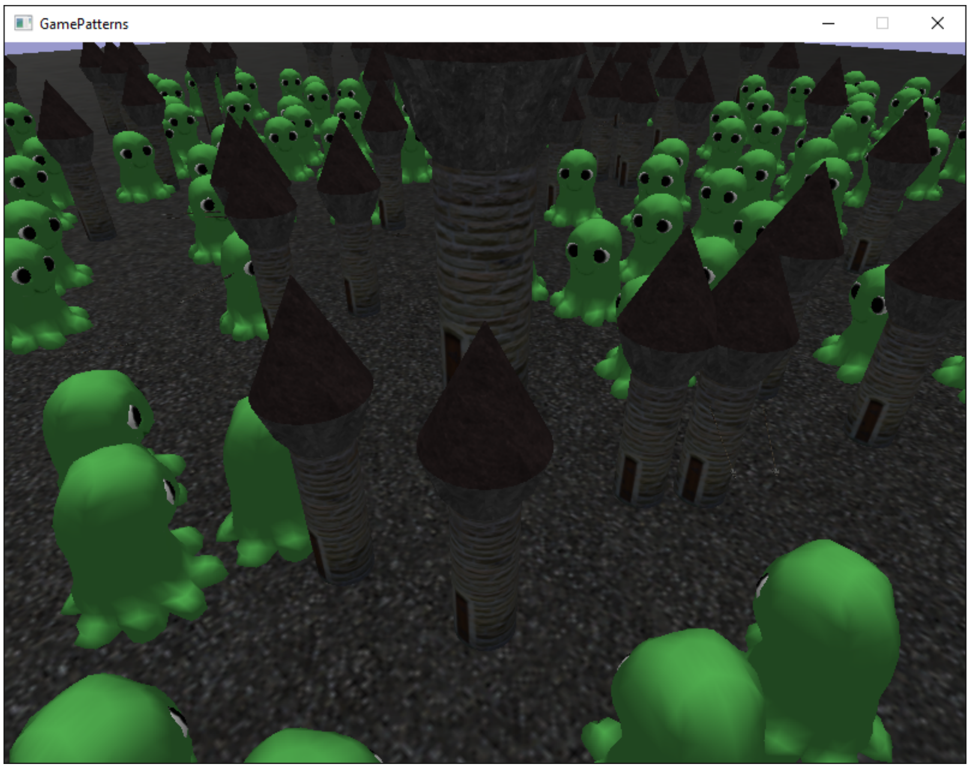

Figure 3.1: Game simulation screenshot

3.3 Base Implementation

In Figure 3.2 a class diagram of the implementation is shown.

Each game object in the game world is derived from the GameObject class. Each game object creates its own model, which has a vector of meshes, for rendering and therefore requires loading a mesh when instantiated. This is an expensive operation since it needs the traversal of hundreds of vertices, UVs and other information. Especially when many objects are deleted and created (like monsters and arrows) this design hurts performance badly.

Subclasses are derived from the tower and monster classes forming a deep inheritance hierarchy. Inheriting classes call in their update function the parent update function and an additional function call to some dummy function, as an extension to the parent class. This should represent different characteristics and behaviors of towers and monsters. In the given class diagram it is also worth noticing that there is a choice from which the class `FlyingGoblinFireWizard` inherits, either from `FlyingGoblinWizard` or

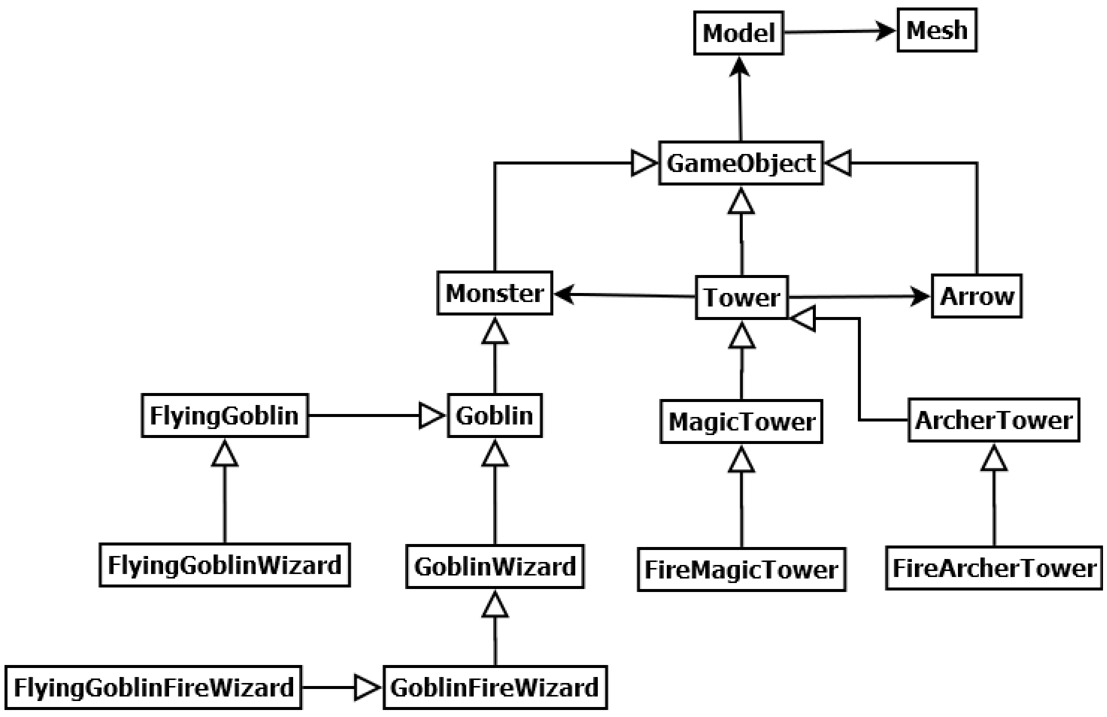

Figure 3.2: Class diagram with inheritance

`GoblinFireWizard`. In this case the class inherits from `GoblinFireWizard` and needs to duplicate the code for the `flying` part of a `FlyingGoblinWizard`.

Each tower manages their arrows in a vector. Every frame a tower iterates over this vector to delete arrows that hit a monster or that are already too far from the tower. Afterwards the tower iterates over a vector of monsters and shoots at the closest reachable monster, adding a new arrow object into its arrow-vector.

The whole implementation can be summarized as in Algorithm 3.1. Though the program seems short and simple it has many flaws concerning performance.

Instances of tower and monster, in particular instances of `FireMagicTower`, `FireArcherTower`, `GoblinFireWizard` and `FlyingGoblinFireWizard`, are stored in vectors at the beginning. For each frame the following is done:

The first loop iterates over all monsters. It checks whether a monster died or has reached its destination. If either is the case it will be removed from the vector and no further computation for the monster is done. Otherwise an inner loop is used to check collision with other monsters. If the monster collides with a monster its moving direction will be altered to push it away from the colliding monster.

The second loop iterates over all towers. Each tower keeps track of its own arrows that has been shot in a vector. The tower instance goes through all arrows and checks if they have

Algorithm 3.1: game simulation

```
init towervector;
init monstervector;
while !quit do
    foreach monster in monstervector do
        if monster reached destination OR monster is killed then
            monstervector.remove(monster);
        else
            var direction = monster.getToGoalDirection();
            foreach monsterOther in monstervector do
                if collides(monster, monsterOther, direction) then
                    breakOutFromLoop;
                end
            end
            monster.move(direction);
        end
    end
    foreach tower in towervector do
        foreach arrow in tower.arrowvector do
            foreach monster in monstervector do
                if collides(arrow, monster, arrowDirection) then
                    monster.health = monster.health - arrow.damage;
                    arrowvector.remove(arrow);
                    breakOutFromLoop;
                end
            end
            if too far away from tower then
                remove arrow;
            end
        end
        tower.shootClosestMonster(monstervector);
    end
    while monstervector.size < MAX_NUM_MONSTER do
        monstervector.addMonster();
    end
    render();
end
```

to be removed. An arrow will be removed if it collides with a monster (again, we need an inner loop to iterate over all monsters) or if it is too far away from the tower. After checking all arrows the tower attacks a nearby monster. The `shootClosestMonster` function iterates over all monsters to find the closest one and shoots an arrow in the direction of the monster, adding a new arrow instance into the arrow-vector of the tower.

When all monsters and towers are updated, new monsters are added again to keep a constant number of monsters in the game world. Finally everything will be rendered.

The collision detection in the sample program uses the Separating Axis Theorem (SAT) algorithm. The SAT algorithm states that two convex shapes do not intersect if there exists an axis on which the projection of the two shapes do not overlap. Otherwise those two shapes intersect [Gre14]. Using the SAT algorithm it is also possible to calculate a minimum translation vector (MTV) with which two objects can be pushed away upon collision. Implementation was taken from [SAT16] and modified slightly.

3.4 Adding the Pattern

3.4.1 Flyweight Pattern

Each time a `GameObject` is instantiated the object needs to load its model from disk. This includes loading the model via the assimp model loader, retrieving the information from assimp's data structure and saving it into the custom Model and Mesh objects for further processing. This takes up a lot of time when new monsters and arrows are spawned over and over again. Since all arrows and monsters look the same it is sufficient to load the corresponding model once and pass the same model as pointer to all instances. In the code only a new constructor for GameObject was added, which takes in a pointer to a model instance (see Algorithm 3.2).

Algorithm 3.2: New GameObject constructor

```
GameObject::GameObject(std::string path) {
    model* = new Model(path);
    // do other stuff
}

//new constructor
GameObject::GameObject(Model* m) : model(m) {
    // do other stuff
}
```

3.4.2 Object Pool Pattern

Another way to prevent costly instantiation is to use an object pool. The maximum number of monsters is known upon application start. Therefore a vector can be reserved for `MAX_NUM_MONSTERS` monsters. Instead of deleting a monster, the monster can be marked as "not in use" with a boolean. However in this particular program, the monster received a new position and full health again, representing a new monster, as at the end of each frame monsters are added again.

On the other hand the number of arrows for each tower is not known beforehand. Upon program start the arrow-vector is an empty vector. If an arrow is needed, first the pool is looked up to check if there is an arrow available. If there is an arrow marked as "not in use" this arrows is used and marked as "in use". Otherwise a new arrow is instantiated, expanding the object pool.

3.4.3 Spatial Partitioning

Collision detection is a very expensive operation. The naive approach, like in the sample program, would be a brute force check with a runtime of $O(n^2)$. With a huge n this operation is unevitable slow. Therefore collision detection is usally separated into two phases, the broadphase and narrowphase collision detection. The broadphase uses less computational intensive methods to find potential colliding objects whereas the narrowphase uses the information from the broadphase to check actual collision using more expensive approaches (for example the SAT algorithm).

For broadphase collision detection there are usually two main approaches, sweep and prune (SaP) and spatial partitioning. The basic idea of SaP is to sort the minimum and maximum dimensions of the axis aligned bounding boxes (AABB) of objects on different axis. After sorting, intersecting AABB are further investigated. SaP uses sorting to cut down the number of collision checks [Gre14]. Spatial partitioning subdivides the game world to reduce the number.

In this experiment performance is improved using the spatial partitioning approach (see Chapter 5). Because the simulation contains many dynamic objects and models that do not extremely vary in sizes a grid is used as a data structure. Using other data structures may cause more overhead for managing moving objects. Each monster is referenced in a cell of the grid depending on its position. When checking for collisions only monsters in the same cell or neighboring cells are considered. When a monster moves the cells are updated accordingly.

Performance gain is varying depending on the cell size and the number of collidables, assuming that all collidables have similar sizes. Cell sizes should be at least as big as the object it contains otherwise it would not be enough to only check for neighboring cells and can lead to more computations. A reasonable cell size is however a size where each cell contains a few collidables and having almost no cells leaving empty. Having a cell size that is too big can result into a runtime near $O(n^2)$ again.

3.4.4 Data Locality

For each operation different attributes of the `GameObject` are needed. For example, the collision detection check needs a bounding polygon and a position. However a `GameObject` has other attributes like rotation and size too which will be loaded into the CPU cache as well. When the scene is rendered the attributes position, rotation and size are needed but not the bounding polygon or a monster's health. In order to have a higher cache hit rate the data has to be reordered.

Algorithm 3.3 shows how `Monster`'s data was reordered. Instead of grouping `Monster`'s attributes together, same attributes are grouped together and contained into one managing class. The three attributes `position`, `rotation` and `size` are grouped together into `Data3d` because they are frequently accessed together.

Algorithm 3.3: Vectorizing attributes

```
class Monster {
    private:
        vec3 position;
        BoundingPolygon transformedBp;
        vec3 rotation;
        bool inUse;
        vec3 size;
        Model* model;
        float health;
        float speed;
        vec3 destination;
}

class MonsterPool {
    private:
        Data3d data3d[MAX_NUM_MONSTER];
        vec3 destination[MAX_NUM_MONSTER];
        float speed[MAX_NUM_MONSTER];
        float health[MAX_NUM_MONSTER];
        Model* models[MAX_NUM_MONSTER];
        BoundingPolygon bp[MAX_NUM_MONSTER];
}
```

3.4.5 Component Pattern

For each inheritance an extra V-Table is kept, when virtual functions are used, and a pointer to the V-Table, while when using composition there is only a pointer to the object. This little overhead might have some effects on performance. In the sample code a part of the inheritance tree was swapped out to use composition. The "is-a" relationship is kept between `GameObject`, `Monster` and `Tower`. Each subclass of `Monster` is also inheriting from `Monster`. The same is applied to `Tower`. However each additional behavior of a monster or tower, for example `FlyingGoblinFireWizard`, is a composition. The `FlyingGoblinFireWizard` is therefore a monster with the component

`Magician`, `FireWielder` and `Flyer`. Figure 3.3 shows the new relations between classes. Using composition the problem of from which class to inherit, introduced in the base implementation, is solved without code duplication.

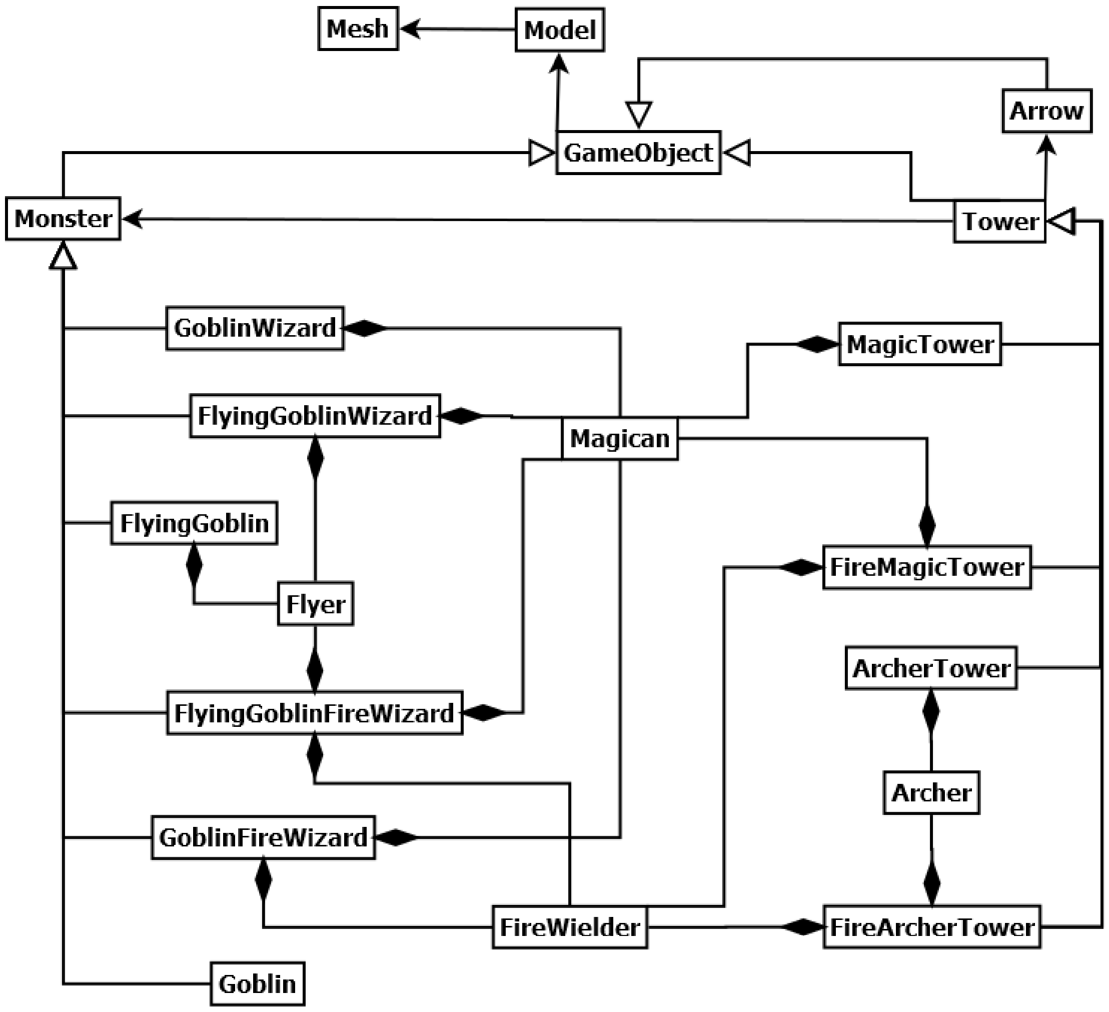

Figure 3.3: Class diagram

CHAPTER 4

Testing

4.1 Test environment

The tests were performed on a HP ProBook 4530s with an Intel(R) Core(TM) i5-2450M CPU (4 x 2.50 GHz). Furthermore the machine was equipped with 4GB of RAM and an AMD Radeon HD 7400M GPU. The operating system was Windows 10 Pro. The program was compiled with the Microsoft C/C++ compiler that came with Visual Studio 2013 Professional (version 12.00.40639.0). For almost all test cases compiler optimization was turned on with the option $/O2$ [Com16]. Only for some test cases concerning the Component Pattern the program was compiled without optimization as there is the possibility that the dummy functions (or even whole classes) were optimized out, influencing the difference in computation time.

4.2 Test method

Upon start the executable is given eight arguments. Five arguments indicate which design patterns should be enabled or disabled. Two other arguments are the number of towers and monsters in the simulation. The final argument is the number of frames that should be rendered. Depending on the arguments for the design patterns different code blocks will be executed, decided by an if-else statement. However not all combinations of design patterns were implemented.

For each test case the needed time for the given number of frames is calculated including profiling information. The time calculation starts after everything has been initialized whereas profiling information include start up initialization and were gathered through sampling. Test cases including Data Locality pattern had additional test cases with instrumentation profiling to measure cache misses. In order to have better comparable results the time step for each simulation step is constant, independent of how long the computation time was for one frame.

A PowerShell script (v. 5) was written to ease testing and profiling. The script starts the Microsoft Visual Studio Profiling tool on the command line (see Algorithm 4.1). For each test case the profiler ran at least five times in order to get a more accurate results.

Algorithm 4.1: Code sniped of the PowerShell script

```
./VSPerfCLREnv.cmd /traceon;
./VSPerf.exe /launch:"pathTo.exe" /file:"outputPath.vspx" /args:"0 0 0 0 0 800
    100 1000" | Out-Null;
./VSPerfCmd.exe /shutdown;
./VSPerfCLREnv /sampleoff;
```

CHAPTER 5

Results

In the following the results are discussed and some profiling information will be provided. All given percentages are relative to the whole execution time and not relative to the calling functions. Percentages relative to calling functions may be given in parenthesis. The pattern names will be abbreviated with FW(FlyWeight), OP(Object Pool), DL(Data Locality), SP(Spatial Partitioning) and CP(Component Pattern)

There were eight combinations of design patterns in total. Table 5.1 shows the first results. For each pattern combination the average needed time in seconds (without initialization time) and frames per seconds (FPS) are given over five iterations. Each iteration calculated 1000 frames using 800 monsters and 100 towers.

FW	OP	DL	SP	CP	Av. time(s)	Av. FPS
0	0	0	0	0	92.49	10.81
1	0	0	0	0	53.14	18.82
0	1	0	0	0	61.76	16.19
1	1	0	0	0	47.97	20.85
1	1	1	0	0	46.53	21.49
1	1	0	1	0	20.27	49.34
1	1	1	1	0	19.14	52.25
1	1	0	0	1	48.01	20.83

Table 5.1: Average times and FPS for 1000 frames, 800 monsters and 100 towers

As to be expected without the patterns the program runs very slow. With an average time of 92 seconds this results in an average FPS of around 11. Total execution time is around 110 seconds, therefore the initialization phase took around 18 seconds. Only 75% of execution time is spend in the actual program. The remaining 25% were spend in other libraries like loading textures and other graphics related tasks. 16% (21% of time in main function) was spend in the GameObject constructor processing models for the object, this includes object initialization before the first frame as well as during frames.

Adding the flyweight pattern improves the program by 43% reducing the overhead of loading models and textures. Total execution time took around 54 seconds, meaning the initialization phase taking less than 1 second. Furthermore 93% was spend in the main function leaving only 7% for graphic related tasks (creating window, drawing to screen, etc). Less than 0.01% was spend in the GameObject constructor, despite instancing new monsters and arrows each frame.

The object pool improves performance as well, however not as good as the flyweight pattern. Using the object pool each object still loads their own models. One reason why the object pool is slower than the flyweight pattern is possibly the loading of the arrow model when no free arrow is found. Furthermore, when spawning a new arrow each tower loops over the list of arrows to find a free arrow introducing extra overhead. Looking into the samples the Arrow constructor had a few hundred samples whereas when using the flyweight pattern there were no samples for the Arrow constructor. Interestingly though, when using the object pool less time was spend in the tower update function, which includes updating arrow position, removing arrows and shooting at monsters, than when using the flyweight pattern.

Using both, flyweight pattern and object pool, the program is 48% faster than the initial program and almost 10% faster than when only using the flyweight. The times spend for the most relevant functions for flyweight/object pool configuration are compared to the flyweight only configuration and are listed in Table 5.2.

	FW (%)	FW (s)	FW & OP (%)	FW & OP (s)
render	3.96	2.10	3.66	1.76
GameObject::collides	67.18	35.70	65.58	31.46
Monster::checkCollision	55.22	29.34	60,54	29.04
Tower::update	18.55	9.86	11,43	5.48

Table 5.2: Times spend in the most important functions

The `GameObject::collides` method checks whether two objects collide. This method is called from within `Monster::checkCollision`, which loops over a vector of monsters for collisions, and `Tower::update`, which loops over a vector of arrows and monsters for collisions. The `render` method is responsible for drawing everything to screen. It seems the most time is saved in the tower's update function when using FW and OP. This observation is in accordance to the previous comparison between FW and OP. Furthermore there is also time saved in the `GameObject::collides` method, which has to be related to the `Tower::update` method as there is no improvement in the `Monster::checkCollision`. The reason for performance improvement in the collision detection is not clear. The number of `GameObject::collides` calls for both configurations are the same, tested with smaller sets of towers and monsters. This eliminates the possibility that there are missing collision checks when using FW and OP. Also, comparing cache misses did not yield clear results.

Adding Data Locality there is a very slight improvement. Instrumentation profiling for counting last level cache misses shows that there were on average 1/8 less cache misses in the `update` function. The `update` function can be split into four further functions: tower update, monster update, arrow update and rendering. For the monster update there were almost no difference concerning cache misses. However the tower and arrow update had together 1/3 less misses. Instrumentation was performed with 80 monsters, 10 towers and 100 frames, as higher numbers would cause too much time consumption and sometimes causing the profiler to stop midway. Further configurations were tested. Table 5.3 compares the non-DL configuration with the DL-configuration with different numbers of towers and monsters. Each test had five iterations, each run calculated 20000 frames.

tower	monster	Av.time	Av.FPS	Av.time(DL)	Av.FPS(DL)	Diff.FPS(%)
800	10	139.79	143.07	141.35	141.49	- 1.58
800	0	130.10	153.73	127.83	156.45	+ 1.74
0	800	761.11	26.28	742.91	26.92	+ 2.38

Table 5.3: Results for DL with different configurations. Average times are given in seconds.

Interestingly when having only few monsters and many towers the performance worsens. However when there were only towers or only monsters performance was improved.

The greatest bottleneck of the simulation was removed with the spatial partitioning. Using the spatial partitioning for the broadphase collision detection doubles the already improved program again. The used grid cell size was 10 x 10 on a 400 x 400 field resulting in a 40 x 40 grid. The monster size is approximately 3 x 3. Changing the grid cell size to 5 x 5, this makes a 80 x 80 grid, and using the same configuration as for Table 5.1 worsens the performance by around 5 FPS. Using a 20 x 20 grid yielded the same results. It seems that a 40 x 40 grid is optimal for the number of monsters size of field. Including DL did improve performance very slightly again.

The component pattern did not improve performance. Since the compiler might have optimized out the dummy functions in general, the program was compiled again, disabling optimization. Different numbers of monsters and towers were used. However the result was the same, neither did it improve nor worsen the performance.

CHAPTER 6

Conclusion

Design patterns are a means to solve reoccurring problems. In this thesis five game design patterns are examined and evaluated concerning performance gain. Except the Component Pattern all other patterns (Flyweight, Object Pool, Data Locality, Spatial Partitioning) did improve performance. Using the configurations as in Table 5.1 performance was improved by 79%, making the program almost five times faster.

One of the two major bottlenecks in the sample application was model loading. Therefore it was no surprise that the Flyweight pattern could almost double the FPS by preventing unnecessary model loading. However in the results it can clearly be deduced that although the Object Pool also prevents unnecessary model loading it had other effects upon adding too. Even when the reason could not be found the results show that even for very small allocations an Object Pool still can improve performance.

The second bottleneck was the collision detection. Spatial partitioning is a very well known technique for broadphase collision detection. In the sample program it was implemented in its simplest form using a grid. This makes the program more than twice as fast as when only using Flyweight and Object Pool.

There was a performance gain by using Data Locality. In [Nys14] the author states that exploiting Data Locality can yield improvement up to 50%. In this case however it did not, which was to be expected. Cache misses were not the reason for bad performance in the beginning. But nevertheless there was an improvement.

Finally the Component Pattern was examined. Unfortunately there was no performance improvement. Neither with compiler optimization nor without. Even with different numbers of towers and monsters there was no difference. It can be concluded that even if composition is faster than inheritance that its effect will be minimal and can be disregarded. But using the Component Pattern has other advantages. It brings flexibility to the code design and helps to write reusable code.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Ass16] *Assimp.* http://assimp.org/, 22-July-2016.

[Com16] *O2 Maximize Speed.* https://msdn.microsoft.com/en-us/library/8f8h5cxt.aspx, 02-August-2016.

[GLE16] *GLEW.* http://glew.sourceforge.net/, 22-July-2016.

[GLM16] *GLM.* http://glm.g-truc.net/0.9.7/index.html, 22-July-2016.

[Gre14] Jason Gregory. *Game Engine Architecture.* A K Peters/CRC Press, 2 edition, 2014.

[Joh94] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. *Design Patterns.* Addison-Wesley Professional, 1994.

[Llo11] Noel Llopis. High-performance programming with data-oriented design. In Eric Lengyel, editor, *Game Engine Gems 2*, pages 251–261. A K Peters, 2011.

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

[Ope16] *OpenGL.* https://www.opengl.org/registry/, 22-July-2016.

[SAT16] *SAT algorithm with MTV.* http://www.codeproject.com/Articles/15573/D-Polygon-Collision-Detection, 02-August-2016.

[SDL16a] *SDL.* https://www.libsdl.org/, 22-July-2016.

[SDL16b] *SDL2 image.* https://www.libsdl.org/projects/SDL_image, 22-July-2016.