Seventh Hungarian Conference on Computer Graphics and Geometry, Budapest, 2014

Optimizing State Changes in Rendering Engines

Dániel Bányász and László Szécsi

Budapest University of Technology and Economics, Budapest, Hungary

Abstract

This paper presents an algorithm for ordering state change operations—including shader context changes and input/output bindings—necessary to render a frame in an interactive application. We expand on the context of the render queue*, but instead of sorting renderable primitives only by material, we propose a flexible framework grouping together drawing calls that share any of the conceivable render states, minimizing the number of CPU-GPU communication instances required to set them.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Graphics systems

1. Introduction

A modern graphics engine has numerous requirements to fulfill. From the consumer viewpoint, it has to create visually appealing images at a constant frame-rate. From the developer viewpoint, it should couple numerous subsystems and allow for flexible addition of new functionality and unrestricted combination of existing features[4]. Game design and content integration should also be easy. Meeting all these criteria simultaneously can be challenging.

Engines are inherently complex software systems that serve as middleware connecting graphics hardware and game logic. Graphics hardware itself has sophisticated architecture running half a dozen shader programs, at least a dozen differently accessed memory constructs, and a good number of fixed-function elements, the operation of which is still highly customizable. Game logic and scene management can be similarly complex, but organized along different principles, and usually not entirely known at the time of engine development. This calls for a software design that is manageable, extendable, and scalable for programmers, even when all the underlying flexibility is exposed. In practice, for programmers with sufficient skills and understanding, programming tasks should not snowball in complexity as the system grows[8].

This level of expandability is not adequately supported by basic instruments of object-oriented design, as encapsulation and inheritance[4]. In order for the addition of new features to happen without compromising the already existing functions in a major way, the engine needs to be very modular. This way changing or completely remaking specific elements of the main engine will not affect other parts of it.

Engines need to support real-time high-fidelity photorealistic visualization of extremely detailed game worlds[11]. Much of the visual experience is dependent on the artistic content, for which a complete pipeline of content creation and integration, with numerous editing tools, is usually supplied. However, it is the rendering capabilities of the engine that define the limits of what game world contents can be. Immersion into a game is reliant on continuous animation, so image frames have to be rendered at consistently high speed[1]. Thus, rendering performance and accommodation of a highly customizable effect range are the two factors that can define the limits of gaming experience.

Unfortunately, flexibility contradicts performance in practice. Both from a programming or a rendering perspective, flexibility translates to a wide and extendable set of features that can be freely combined in the definition of game world objects. Optimization for performance, on the other hand, means the elimination of multiple executions of the same task through exploiting uniformity. Consequently, almost every optimization technique depends on restricting what the engine can do. For example, grouping some of the game object properties into *materials*, then drawing objects sharing the same material together is a natural and widely accepted technique. However, what exactly constitutes a material depends on design decisions, where choosing a too narrow subset results in little reuse, while a too wide subset either leads to too much uniformity or too many materials. Multi-level material systems could address this problem, but then the hierarchy of material parameters already assumes a lot about their usage. Even if this is based on well-established best practice, it is a restriction imposed on how the content should be prepared for optimal performance.

In this paper we describe the outlines of a flexible graphics engine, focusing on the capabilities of optimized rendering without assuming any pre-established classification, uniformity or hierarchy of game world objects. This is to provide the means of high-end rendering quality while maintaining complete freedom in the design of material or shader systems and game object construction. The main contribution of the paper is a run-time algorithm to order all visible game world objects for rendering so that state change overheads are minimal.

Our optimization algorithm is a greedy one, which means it is not always optimal. Greater performance increase can be achieved using batching of elements or instancing. However, this algorithm can easily work with batched resources and instancing, is independent from the concrete graphics API and development environment and can easily be modified and expanded to the needs of any certain engine.

2. Component-based engine architecture

The choice of a *game object model* or *entity model* is a defining feature of game systems[4]. Small games fare well with the classic *monolithic model* that relies on object-oriented inheritance hierarchy. Every different type of entity is implemented in a class, with common features extracted into superclasses. This would imply multiple inheritance where entity types share features along multiple taxonomies. Due to the inherent design challenges of multiple inheritance, it is only used with restrictions. Interfaces and *mix-in* classes add some flexibility to a monolithic inheritance tree.

The most severe issue with an inheritance based game object model is that of adaptability and extensibility. Game development is often an organic process where technological features may inspire gameplay elements[4]. If a new feature is added to an entity class, it is very likely that sooner or later other, formerly unrelated entities could also use the feature. Moving shared functionality to the lowest common base class results in bloated classes at the top of the inheritance hierarchy, and requires switches to turn off functionality where not needed. This defeats the purpose of object-oriented decomposition. Mix-ins solve this problem only for simple bits of functionality that do not require further decomposition.

Another, related problem is the strong coupling of interfaces. Whenever an entity class requires that the interface is changed, this can affect a complete subtree of the inheritance hierarchy.

The need for integration of several subsystems in an engine rapidly and naturally leads to component-based systems, where entities are defined as collections of components. The idea lends itself to implementations ranging in complexity and flexibility from simple *composition over inheritance* schemes to run-time configurable object compositions. *Property-based* systems can be seen as an extreme case, where all entities are disassembled into atomic components, and behavior is no longer defined by components themselves, but the logic linking those components. Operations are dispatched not based on the type of one component, but on the occurrence of certain sets of components.

The decomposition of game objects into components can easily be extended to features that are parts of virtual worlds, but not readily viewed as game objects in simple systems. Light sources, cameras, trigger zones, and audio sources may very well share functionality with renderable objects.

A component-based architecture allows simple, yet powerful and flexible couplings between entity aspects and the ability to create or change these connections very easily. The more abstract the component management is, the more machinery it requires to glue a component into the system, but also the less existing components constrain what new components can do.

3. Testbed implementation

In our implementation we opted for a strong and strict decoupling of subsystem functionalities into rendering, physics, etc. components. Variations within those components are handled by inheritance, e.g. there is a common interface for physics components allowing creation, destruction, and simulation. However, how simulation results are transferred to rendering components is not defined in either physics or rendering components: they are independent. Also, there is no entity class storing components that belong to the same entity, it is only the components themselves that store an entity identifier. Entities exist only as components sharing the same identifier.

As in the rendering and physics components example, components have to communicate, and they need to do so without compromising loose coupling. We need an object in the engine that knows about all the different types of components and can invoke the specific behaviors from these elements. These objects are *visitors*, and they implement the Visitor Pattern[7, 6]. For every action (render, update, release resources etc.) we create a specific visitor object that knows how to handle given components. This way the knowledge about the concrete coupling and the communication between components is comprised in the visitors and not scattered throughout the engine. Implementing visitors is not substantially different from implementing virtual functions in a monolithic inheritance engine, but visitors can operate on multiple classes.

4. Dependency injection

This approach incurs some communication overhead, as visitors need to look up objects based on entity identifiers and discern their types, but this is easily acceptable for game logic resolution. When it comes to rendering, however, these loose connections can be a burden. In a monolithic engine, all data for drawing an entity would be encapsulated in one class, and a single render method would be responsible for drawing the entity. With components, the visitor must gather all the parameters, variables and resources necessary.

This overhead can be eliminated if connections between critical components are made more direct. In order to achieve this without undermining component separation, we employ the *dependency injection* pattern[3, 9]. This inverts the control flow between various components supplying rendering parameters (e.g. model, camera, light poses) and the component performing rendering. While originally the render-performing component should locate and query all other components, with dependency injection it is the other components that register themselves.

5. Shader components

The render-performing component in our implementation is called the *shader component*. None of the data requirements for a shader are hard-coded. Instead, they are acquired runtime using *shader reflection* API[13]. After loading a shader program, all of its parameter and resource binding points, along with their names used in the shader programs, are gathered and saved. A shader component stores every parameter by name. Visitors can connect data providers and shader components by querying provided items and setting references to them into named shader parameters. A visitor might contain the logic linking data items to names, but if shader program authoring is performed at the same time as component design, it is a clearer solution to have the provider components use the same naming convention as the shader programs.

After the shader components have been populated with appropriate settings, rendering does not require any communication between components. As data provider components pass data references to shader components, no synchronization is necessary when the data changes.

6. Related work

A high number of render components share a large subset of parameters. The camera properties are the same for all of the objects in the frame, texture samplers can also be common between lots of elements, and the lights can affect groups of objects in the virtual world. We may be drawing a lot of entities with the same texture or with the same world matrix, if one object consists of multiple render components. What we need to do is find the overlapping resources and set them only once for a large number of objects, minimizing the state changes and the communication between CPU and GPU, maximizing the utilization of the PCI-Express bandwidth, and therefore optimizing the rendering process.

The most efficient way of reducing CPU-GPU communication is reducing the number of draw calls, or *batches*, themselves[12]. This can be done if several entities can be drawn at once. As no state changes during the processing of a batch are possible, any variation in drawn geometry must be accommodated for dynamically in shaders. This requires pre-transformed static objects stored in common geometry buffers, the use of texture arrays instead of switching textures, and generic shaders accommodating different material models. These optimizations can be performed when designing the shader library and the scene management for the engine.

A special, hardware-supported case of batching is *geometry instancing*[5]. This allows rendering the same geometry several times, with shaders able to process them with different parameters.

For the purposes of run-time performance optimization, we assume batching has already been done, and the entities may already constitute several game objects batched together. In other world, we define an entity as what can be drawn in a single batch.

Minimizing the number of state changes can be accomplished by ordering batches so that those using similar state settings are performed one after the other. The tool for this is the *render queue*[2], which accumulates batches, and executes them only after ordering. This solution is employed generally, but the ordering strategy is often arbitrary, or manually tuned to a certain purpose. Typically, engines have a material system, where materials encompass a subset of all render states. How this subset is chosen is based on industry best practices, but it is not adaptive on the momentary runtime set of entities. The batches are ordered according to material IDs, which are statically assigned.

Our algorithm optimizes draw calls not by materials but by used states and resources. We do not differentiate between render state defined at material or entity level, and dynamically adjust the ordering strategy depending on the composition of entities in the scene.

7. The rendering optimization problem

The virtual world, or *scene*, is a collection of constituent *entities*. From the rendering point of view, we consider those entities that are *renderable*, meaning that they can be drawn into a render target image. For each of these entities there exists a set of various *render components* that define the rendering behavior. As a simple, but typical case, these could be a *pixel shader component*, a *vertex shader component* and an *indexed geometry component*. Such a setup allows multimaterial meshes if they are instantiated as several single-shader entities.

The objective is to determine how and when to upload or bind data to the GPU in order to minimize CPU-GPU data transfer and communication latency. Several entities need to be rendered using the same resources, shader programs and state blocks, and thus, re-binding—or even re-uploading—these could needlessly stall operation. For simple scenarios, this could be addressed manually in the content creation phase, using a material system that orders materials so that similar ones follow each other. Then, entities can be sorted by material ID in a render queue, in run-time, before rendering. However, content-creation-phase ordering might not be optimal for the actual set of entities that are present at run-time. Furthermore, if rendering behavior is not merely defined by a material, but by an ever extendable set of components, then the render queue approach is insufficient. For example, a future gameplay decision may dictate that the player is given a gun that changes the texture of entities fired upon. This can easily be solved by creating a new render component class that imposes a texture setting when visited, overloading the material default. However, this could break the material system and render queue ordering, calling for a custom workaround to be implemented. In order to avoid development overheads of this kind, we aim to determine the optimal render ordering at run-time.

An exact method can be easily imagined using a graph, every node of which is an entity. The graph is a complete graph, and going from one node to another means rendering one entity after the other. This graph is also a weighted one, where the edge weights between nodes represent the severity of state-changes between GPU states when rendering one entity after the other. Before we render an object, we bind (or even need to upload) all the necessary resources, set the render states, and when we are finished with the rendering, we have to update some or even all of the parameters in order to prepare for the drawing of the next object. When the weight of an edge is low, it means we barely have to change anything. When this number is high the number of differences between used resources is significant, meaning we need a lot of data transferred to the GPU before the actual rendering begins. The edge weights should also represent the differences between the performance impacts of the API calls.

After the graph is ready, we need to find a path that includes every node once and only once, and has minimal weight, i.e. a minimal weight Hamiltonian path. This would mean that we have rendered every entity with minimal or no overhead. Finding a Hamiltonian path in a complete graph is trivial, but finding one with minimal weight (known as the *travelling salesman* problem) is known to be NP-hard[10]. There are various fast approximate algorithms, however. Our solution can be seen as another such algorithm, also exploiting our knowledge about the structure of the edge weights.

8. The algorithm

There is a high number of configuration items that influence GPU operation. These include render states, input/output resource bindings, dynamic resource contents, shader programs or dynamic shader linking parameters. For the sake of brevity, we will refer to all of these as render states, even though setting costs and the implied amount of CPU-GPU data transfer might vary greatly for different types.

Render states are set using graphics API methods. An atomically settable render state is something that can be changed by invoking a single method. The basic intuition of our algorithm is that if there are two atomically settable render states with similar setting costs, one of which takes just a few different values in the course of rendering, but the other one is likely to be different for all entities, then it is more beneficial to order entities according to the first one.

To explain this, let us assume we need to order n entities. Let M be one of the graphics API methods, associated with a render state. The cost of invoking M is c_M, and there are v_M different values of the render state used. Then, the cost for invoking M individually for every entity is $c_{\text{base}} = c_M n$. If sorted by the values of the render state, the cost will be $c_{\text{opt}} = c_M v_M$. The *gain* is $c_{\text{base}} - c_{\text{opt}} = c_M(n - v_M)$. Thus, the gain is more significant if there are fewer possible values or if setting the state is more expensive.

Thus, our algorithm will find render states that are used by a high number of entities and are expensive to change. We set these early and keep the GPU and its memory in the correct state while all corresponding entities are drawn. To achieve this, we count the number of different parameter sets for every graphics API method that causes data transfer or state change. We refer to a method call with a certain set of parameters as a *task*. Two calls with the same set of parameters are not considered separate tasks. If a method is invoked by numerous entity components, but repeating only a few tasks, then a high number of entities share the render state setting.

Our algorithm first selects a render state, ordering along which offers the maximum gain. For this we need to know the number of different parametrizations the methods setting render states are called with. In other words, this is the number of tasks pertaining to the method. Also required is the cost of invoking a method over the API. When the maximum gain render state has been selected, the entities are grouped by the value of the render state they require for rendering—in other words, which task of the method they invoked. The resulting groups consist of entities for which the render state needs only be set once, if the group is rendered together. Such a group of entities can further be divided choosing another render state (i.e. API method) to discriminate by. This can be repeated recursively, until there remains no other API method but the final draw call, which has to happen for every entity.

8.1. Optimizer

The central construct of our algorithm is the *optimizer*. This provides an interface equivalent to that of the graphics library used to control GPU operation, but for all method calls, the calling entity also has to be specified. The optimizer does not immediately execute the graphics library method calls, but records them, and generates an optimized list of *commands*, that can be played back to the same effect.

Thus, all shader objects and other rendering components of entities issue parametrized calls to the optimizer, instead of the graphics API. In order to explain the optimizer, let us introduce the following nomenclature:

- A *method* is a graphics API function exposed on the optimizer's interface.
- The *method cost* is the overhead incurred by calling the graphics API function.
- A *task* is a graphics API call with a given parametrization, i.e. a method–parameter-set pair. Invocations of the same method with identical parameters are not considered separate tasks.
- The *callers of a task* are those entities, the components of which have invoked the task's method with the task's parameter list.
- An *entity set* is a set of entities to be drawn.
- A *bin* is a set of the tasks that refer to the same method.
- A *method of the bin* is the method all tasks of the bin refer to.
- The *bin gain* is the reduction in state setting overhead achievable by grouping entities according to their tasks in the bin. It is computed as the number of entities minus the number of tasks in the bin, weighted by the the method cost.
- A *command* represents one execution of a task. Thus, it is a wrapper for a graphics API call with a given parameter set. By extension, API draw calls can also be wrapped in commands.
- The *command list* is a replayable sequence of API calls.

The optimizer stores all tasks into bins, one for every method. We refer to the bin that has the highest gain as the *best bin*—the choice is arbitrary in case of a tie. For every task, the parameters and the list of callers are also stored.

The optimizer maintains this data structure when receiving invocations from rendering components of entities. If a method is called with a parameter set that differs from every previous, a new task is created, and added into the appropriate bin. If the parameter set used by the calling entity was already associated with a task, we just register the entity as one of its callers.

Draw calls—the graphics API function calls that initiate pipeline operation with current bindings—are not wrapped or stored as tasks. These cannot be eliminated by ordering, as a draw call must happen for every entity. (As discussed in Section 6, optimizing the number of draw calls using instancing or batching is out of the scope of our investigation.) On the other hand, the output optimized render list must contain draw commands. Thus, the optimizer maintains an entity set, containing all entities that have to be drawn. This set is identical to the union of callers of all tasks, and thus redundant in information to the task bins. However, it is much more useful for entity set partitioning operations.

8.2. Optimization

When entities are created, rendering components are also instantiated. The core functionality is encapsulated in the shader component class. Shader components manage all program and resource bindings of the GPU pipeline. Initially, however, shader components are devoid of bindings, and it is the role of the visitors to fill these as dictated by other components. Thus, components are decoupled, and linked only by visitor logic. During the process, shader components receive memory locations for all the shader variables' desired values. After these bindings are made, the engine can start creating the optimized render list.

First, the optimizer is instantiated, with empty bins for all methods, and an empty entity set.

Then, we emulate the rendering process, setting required resource bindings and render states, and rendering the geometry through the optimizer. This is accomplished by visiting the shader components, that inform the optimizer about required settings, per entity. The optimizer receives these calls, and creates, bins and updates tasks as described in Section 8.1. Also, the caller is inserted into the entity set, if it was not a member yet.

By the end of this step, the optimizer has been initialized, and the OPTIMIZE function (Algorithm 1) can be called with the set of all bins and the complete entity set as inputs, returning a list of commands (see Figure 2 for an overview).

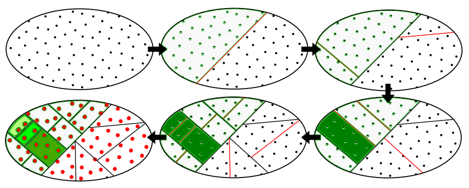

Figure 1: *Recursive subdivision of the render state space.*

The function partitions the entity set into groups recursively along a set of methods (Figure 1), represented by the bins of their tasks. For an empty group E, there is no action required (lines 2-4). Otherwise, the list of commands to render the group must be returned, assuming all appropriate settings have already been applied, apart from those using the methods of $\mathcal{B}$, which are currently subject to optimization. The necessary commands are appended to an initially

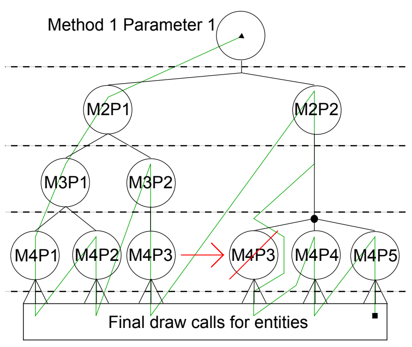

Figure 2: *The algorithm recursively partitions entity sets along tasks, implicitly creating a tree. The green polyline indicates the order in which the tree is traversed to find the output render command list.*

Algorithm 1 Optimize function

```
 1: function OPTIMIZE(set of bins $\mathcal{B}$, set of entities $E$)
 2:     if $E$ is empty then
 3:         return empty list of commands
 4:     end if
 5:     $R \leftarrow$ empty list of commands
 6:     if $\mathcal{B}$ is empty then
 7:         for each entity $e$ in $E$ do
 8:             $R \leftarrow R +$ new command(draw $e$)
 9:         end for
10:     else
11:         $M \leftarrow$ best bin in $\mathcal{B}$
12:         for each task $p$ in $M$ do
13:             $R \leftarrow R +$ new command($p$)
14:             $I \leftarrow \{$callers of $p\} \cap E$
15:             $\mathcal{D} \leftarrow$ empty set of bins
16:             for each bin $B$ in $(\mathcal{B} \setminus M)$ do
17:                 $\mathcal{D} \leftarrow \mathcal{D} \cup \{q | q \in B \wedge$ (caller of $q) \in I\}$
18:             end for
19:             $R \leftarrow R +$ OPTIMIZE($\mathcal{D}, I$)
20:         end for
21:     end if
22:     return $R$
23: end function
```

empty list R (line 5). We refer to adding commands performing tasks into the render list as *compiling* the tasks. If the bin set $\mathcal{B}$ is empty, then all settings are already compiled into the command list, and only the draw calls for all entities must be issued (lines 6-9). Otherwise, a best bin M can be selected (line 11).

Then, we loop over all tasks in the best bin (lines 12-20). Processing one task should compile the setting of a render state and the rendering of all entities using it. Therefore, we append the command executing the task p to the command list R (line 13). Then, all the entities that use p are selected into set I (line 14). These entities must be compiled, but without calling the method of M, as that has already been properly applied. This is accomplished by calling OPTIMIZE with entity set I, and a reduced set of bins $\mathcal{D}$ (line 19). The returned command list is appended to R.

Set $\mathcal{D}$ is populated from bins of $\mathcal{B}$ other than M, copying all tasks that belong to an entity in I (lines 15-18). Note that bin M has been eliminated from the recursion, because the entities we are left with share the same setting with respect to that method. Tasks that concern entities other than the ones in the processed group are also eliminated, so that the gain computation for the bins remains correct.

After all tasks of M have been processed, the command list is complete and can be returned (line 22).

The top level call of OPTIMIZE returns a list of render commands that is as close to the optimum as the greedy approach allows.

However, it is possible that there is some unnecessary duplication amongst method calls. Suppose we render three entities, using two methods A and B, issuing tasks a_1, a_2, b_1 and b_2. The three entities e_1, e_2, e_3 use settings (a_1, b_1), (a_1, b_2) and (a_2, b_2), respectively. Partitioning by A separates the first two entities, resulting in the command sequence (set a_1, set b_1, draw e_1, set b_2, draw e_2). On the other branch, we get (set a_2, set b_2, draw e_3), where (set b_2) is superfluous. This suboptimal behavior is possible because separate branches are processed independently, and no considerations for minimizing state changes between branches are taken. While this concession has to be made in favor of the fast greedy evaluation scheme, a trivial elimination of ineffective commands can improve our solution in simple cases like the one presented above. We could see these as lucky coincidences, where the rendering of one group of entities happens to end in a render state in which rendering the next group should start—but these lucky coincidences are quite probable if only a few possible states exist.

Visiting rendering components again is unnecessary, as the render command list contains every method call with a full parameter list, with pointers to resources originally specified by the render components. Thus, no additional overhead other than the execution of the tasks is incurred when the command list is played back for rendering. If no entities are created or destroyed, or their components otherwise altered, the command list does not need to be re-generated in every frame. Dynamically uploaded data, like the contents of constant buffers storing camera and model transformations is allowed to change as long as the data resides at the same memory location. Thus, even for a dynamic scene, every frame can be rendered by playing back an optimized sequence of graphics API calls.

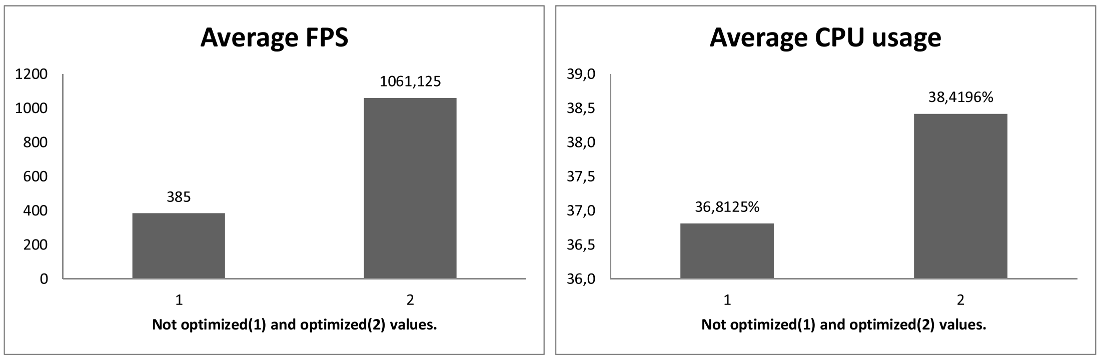

Figure 3: *Optimized CPU-GPU communication resulted in higher render speeds and increased CPU utilization.*

9. Results

We tested our algorithm on an Intel Core i5 3570K CPU with 8 GB RAM and an MSI N670 PE 2GD5/OC graphics card. With only a small initial computational overhead of creating the render lists, the speed of the actual rendering increased significantly. The differences between the optimized and unoptimized runs were apparent on the test machine.

These tests were conducted using a simple and carefully constructed virtual scene (see Figure 4). A small number of objects were placed in the world, some of them forming optimizable groups by using several common resources. With a test scene this size, the algorithm was fully observable.

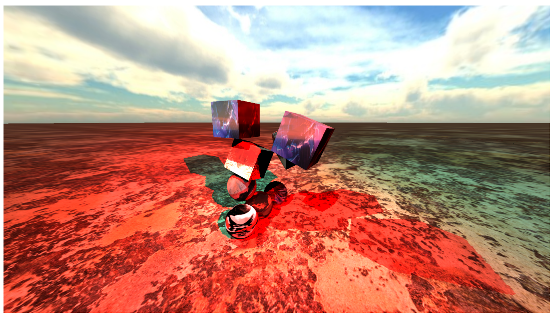

Figure 4: *Test scene with deferred shading and depth map shadows.*

We compared two methods of drawing this virtual world. One was with the suboptimal rendering, which draws entity by entity, collecting components and setting parameters individually. The other was the optimized method which uses the render list created by the algorithm. There are multiple light sources casting shadows using shadow maps, so each of them has its own render pipeline, meaning they all have to render the scene from their point of view. We use deferred shading, so each of the lights contributions are blended together in a final draw sequence. In conclusion, the scene will be rendered multiple times using separate render queues, separate optimized passes, each pass manifested in their own optimized method list.

The engine has some built-in tools to measure specific aspects of performance. We created the scene and the engine environment in order to quantify the gain of using our optimizer algorithm. In the first test, we compared the optimal and the suboptimal renders with no other restrictions on the engine, meaning we disabled V-Sync and any other frame rate limits. This way the GPU ran at 100% capacity generating as many frames a second as it possibly could. We ran the engine for a few seconds measuring at a steady interval the CPU usage and the rendered frames per second count. The numbers showed that using the optimized method list to draw the scene generated nearly three times the frames a second, than rendering without the optimized list (Figure 3).

The CPU usage was slightly higher during the optimized rendering. This can be explained by the fact that without optimization, the GPU communication overhead blocks the CPU from getting higher amount of work done. During the tests, the engine had nothing else to do but to render. It processed some messages from the operating system in order to allow the user to move the camera and close the program window, but apart from that, it only prepares the scene and uploads the data to the GPU. Faster render means more CPU work, because the engine will require more work to be done for the rendering process from the CPU side.

In an effort to determine the accurate gain in CPU power we tried to level the playing field in terms of CPU usage, and framelimited the engine. So at the second round of measurements, the FPS was capped at 60. This produced very low CPU load, probably because of the small size of the virtual space. The CPU usage while rendering from the optimized list however is nearly half with this simple scene than when we render the entities one by one (Figure 5). Hopefully even with a more complex virtual world this gain will have a huge impact on the overall performance.

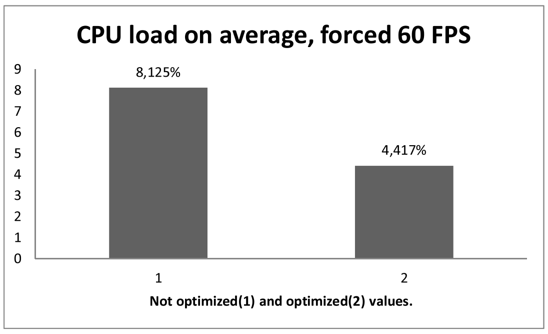

Figure 5: *CPU usage for the optimized and non-optimized cases over several frames with limited frame rate.*

10. Future work

The proposed solution needs further verification for more extensive scenes and full, plausible gaming scenarios. More quantitative comparison with material-based render queues are required, and a study of circumstances where the theoretical merits of the proposed algorithm manifest most strongly. We need to investigate further on the issue of when the render command list needs to be rebuilt, and whether it is possible and advisable to keep parts of the list and update others, trading optimality for cheaper list updates. In a related issue, the effect of visibility algorithms removing and adding entities for the set to be drawn should be addressed. We have not yet investigated another important option for increasing rendering performance, which is sorting entities by approximate depth. We plan to examine whether this sorting strategy can be combined with our optimization algorithm. Also, it would be useful to know when is one or the other more beneficial.

11. Acknowledgements

This work has been supported by OTKA PD-104710 and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

1. Kajal T Claypool and Mark Claypool. On frame rate and player performance in first person shooter games. *Multimedia systems*, 13(1):3–17, 2007.
2. Christer Ericson. Order your graphics draw calls around! `http://realtimecollisiondetection.net/blog/?p=86`, 2008.
3. Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, 2004.
4. Jason Gregory, Jeff Lander, and Matt Whiting. *Game engine architecture*. AK Peters, 2009.
5. Emmett Kilgariff and Randima Fernando. The geforce 6 series gpu architecture. In *ACM SIGGRAPH 2005 Courses*, page 29. ACM, 2005.
6. Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
7. Bertrand Meyer and Karine Arnout. Componentization: the visitor example. *Computer*, 39(7):23–30, 2006.
8. Bob Nystrom. *Game programming patterns*. 2013.
9. Erick B Passos, Jonhnny Weslley S Sousa, Esteban Walter Gonzales Clua, Anselmo Montenegro, and Leonardo Murta. Smart composition of game objects using dependency injection. *Computers in Entertainment (CIE)*, 7(4):53, 2009.
10. Gerhard Reinelt. *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, 1994.
11. L. Szirmay-Kalos, L. Szécsi, and M. Sbert. *GPU-Based Techniques for Global Illumination Effects*. Morgan and Claypool Publishers, San Rafael, USA, 2008.
12. Matthias Wloka. Batch, batch, batch: What does it really mean? In *Game developers conference*, 2003.
13. Jason Zink. Direct3d 11 shader reflection interface. `members.gamedev.net/JasonZ/Heiroglyph/D3D11ShaderReflection.pdf`, 2009.