universität
wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

"Reducing CPU overhead for increased real time rendering performance"

verfasst von / submitted by

Daniel Martinek BSc

angestrebter Akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2016 / Vienna 2016

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik UG2002
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

For some years the performance of graphics processing units (GPUs) has increased very rapidly. A simple way to compare computing power is the count of floating-point operations per second (FLOPS) that the hardware is able to issue. Although this measurement is by no means the only indicator that has to be taken into account for performance of real world computing devices, it allows us to easily compare the baseline performance of processing units. In 2005 high end consumer GPUs[1] were able to compute 7.2 GFLOPS. Ten years later in 2015 a high end GPU[2] is capable of a thousand times the processing power with 7.2 TFLOPS.

In contrast to the improvements in the hardware, the application programming interfaces (APIs) currently in use are still partly based on the hardware design from 10 years ago. The development of GPU hardware has increasingly moved from fixed graphics functionality to a more general purpose architecture. Some of these changes have been accommodated by the sporadic addition of features like compute shaders but are still not fully reflected in the APIs.

Besides not exposing the whole functionality of the hardware, APIs like Direct3D 11 execute heavy validation and memory management operations in the background. As shown in [8] this drastically reduces performance when calling API functions too often. The CPU thread that commences these operations will effectively limit the number of objects that can be drawn per frame.

An alternative to dedicated graphics APIs are so called general purpose graphics processing unit (GPGPU) APIs. In contrast to Direct3D or OpenGL these APIs do expose the generality of modern GPUs. They however don't expose the graphics pipeline parts of the GPU like the rasterization or texture units, making the interfaces a poor choice for graphics heavy development.

Multiple new graphics APIs[3][4] have already been proposed to improve the situation. Unfortunately they are either not yet available or only supported on a small subset of personal computing devices.

[1] GeForce 6800 Ultra Extreme
[2] Radeon R9 Fury
[3] Vulkan https://www.khronos.org/vulkan
[4] D3D 12 https://msdn.microsoft.com/en-us/library/windows/desktop/dn899228(v=vs.85).aspx

1.2 Outline

In the course of this master thesis a rendering algorithm is described that is able to speed up rendering for a range of fully dynamic scenes by reducing API overhead. The proposed architecture works around current API limitations by making heavy use of the small set of general purpose functionality provided by the Direct3D 11 graphics API. Compute shaders are used to combine all rendered geometry into a single batch for drawing. The proposed changes are compatible with all widespread shading techniques like deferred rendering, deferred lighting or forward rendering.

A custom rendering engine, called *breach*, was created to implement both the proposed rendering architecture and a basic rendering architecture based on one draw call per object. This engine is then used to measure the relative drawing performance in multiple scenes.

2 Introduction to real-time rendering

Video games, virtual reality, three dimensional navigation apps, websites for interactive product customization, all of these applications have at least one thing in common: they display a virtual scene and let the user interactively manipulate it. To achieve interactivity in a computer application the time from user input to application output is, as shown by Miller in [19], restricted to a value below 0.1 seconds. In the case of fast paced computer games this time span or latency can be as small as 33 to 66 milliseconds. This value covers the whole available time for computations ranging from filtering and handling of user input, over computing reactions based on the input (e.g. physical movement or artificial intelligence), to the *rendering* and display of the final image. The desired latency in combination with the need for smooth animation therefore limits the amount of time available to generate the image to about 16 to 33 milliseconds per frame.

2.1 Using a graphics API

In modern real-time computer graphics most of the computations necessary for rendering an image are carried out on a co-processor called the graphics processing unit (GPU). Since the development of the first GPUs to now they have evolved from very specific processors with fixed functionality to extremely wide general purpose SIMD processors. In the early years of special purpose graphics hardware the hardware dictated very much the functionality and implementation of the graphics pipeline. Figure 1 shows the graphics pipeline of Direct3D 7 as an example for a fixed function pipeline. The only way a graphics programmer was able to steer the outcome of the pipeline was to change the parameters of the fixed function blocks.

The pipeline of Direct3D 11, shown in Figure 2, offers a more modern approach to the graphics pipeline. For a big part it is however still based on the same principles as the fixed function pipelines. A fixed function rasterizer exists that turns triangles into pixel fragments, as well as an output merger stage that combines pixel values. To transport and transform the data between the fixed blocks new shader stages were added. These shader stages contain small programs, the eponymous shader. Based on the position in the pipeline each shader has a special task. The *vertex shader* is for example responsible for projecting every point describing the polygonal representation of a three dimensional model onto the two dimensional screen. A *pixel shader* on the other hand is invoked for every pixel fragment that is generated by

the rasterizer and computes the final color of the pixel that should be output to the screen. In addition to these special purpose shaders, a general purpose *compute shader* exists that can be used to carry out calculations without dependencies on the fixed function stages of the pipeline.

The programmability of the hardware allows graphics programmers to find new ways to utilize the hardware to improve the performance or quality of real-time rendering applications by using alternative rendering algorithms.

To send a geometrical object or mesh through this graphics pipeline a *draw call* is issued. This call will instruct the GPU to use all the resources currently bound to the pipeline to render a new object. These resources are:

Vertex Buffer

This resource contains all data that is associated with the vertex points defining the mesh e.g. position, normals, texture coordinates, etc ...

Index Buffer

The index buffer consists of a list of indices into the vertex buffer. Depending on the primitives that are rendered (points, lines, triangles, etc ...) a number of consecutive indices define the vertices used to generate a specific primitive.

Shader

A shader is a small computer program running on the GPU at different fixed stages.

Constant Buffer

This kind of data buffer is used to supply a small amount of data to the shader program that is constant per draw call. As the size of this buffer is small, it can be stored in very fast memory and is often used to supply a shader with object data like color or transformation matrices.

Texture Buffer

Textures are multi-dimensional buffers (mostly two dimensional) supplying arbitrary data to the shader. The interpretation of the texture data is up to the shader but often textures are used to add details to meshes that are too small to be efficiently depicted as geometry.

Structured Buffer

This buffer type is designed to hold arbitrary structured data that is too big to be stored in a constant buffer and has no need for the additional functionality of a texture buffer like texture sampling.

These resources are bound to the pipeline using functions supplied by the graphics API. As described in [10] this incurs a cost depending on the type of operation done. The API has to validate the call on the CPU side to check if the resource bound is of the correct format, is ready to be used, and not used by other pipeline stages that would conflict with the expected usage. On the GPU side resources that are needed could be still in use elsewhere in the pipeline. If no other operations can be interleaved the pipeline will be stalled until the resource becomes available.

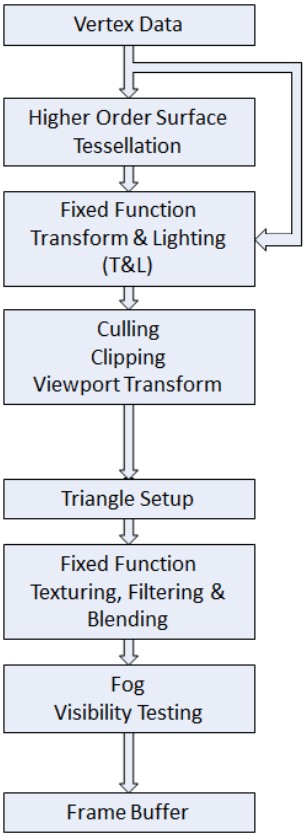

Figure 1: DirectX7 fixed function rendering pipeline.

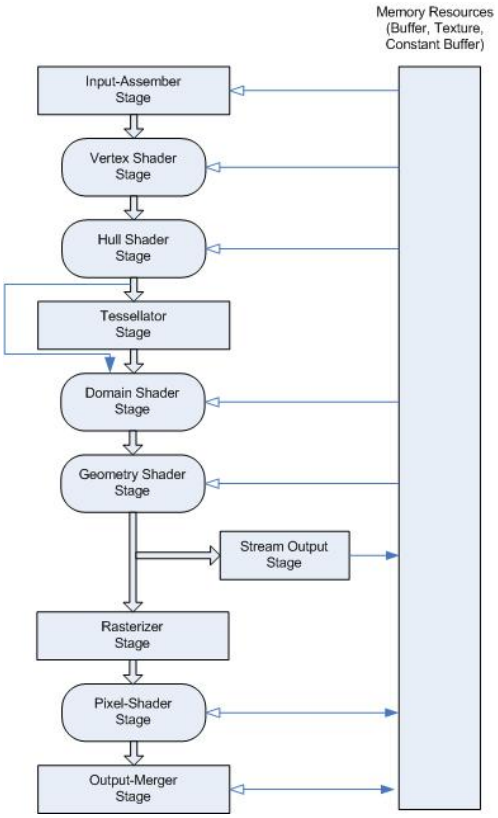

Figure 2: Direct3D 11 programmable rendering pipeline. [6]

[6]Image taken from Microsoft Direct3D 11 documentation.

2.2 API future

The last years mark a turning point for graphics APIs. In 2013 AMD presented the Mantle API [1] as a low level alternative to OpenGL and Direct3D. Mantle was developed in cooperation with multiple game development and graphics companies to allow a more hardware-centric approach to graphics programming. The API and runtime was simplified by moving complex functionality like concurrency and memory management from the runtime to the user application. This step allows drivers to be potentially more reliable and faster, but also increases the workload of the application developer. In 2015 Mantle development was halted in favor of Vulkan.

Although the API was maintained for a very short time only and saw no widespread usage it lead to some important changes. On desktop computers the two APIs OpenGL and Direct3D are widely used and both are currently in the process of being succeeded by new versions: Vulkan and Direct3D 12. The Vulkan API [16] is the spiritual successor of OpenGL. It is developed by the Khronos Group and is partly based on the Mantle API. Direct3D 12 [7] is similar to Vulkan a complete redesign of the API. To ease transition a special 11on12-Layer was developed that allows users to utilize the Direct3D 11 API functions with the Direct3D 12 runtime.

Both APIs strife to simplify the graphics driver and API runtime by moving more work into the user application. Both concurrency and memory management have to be implemented on the application side. From a performance standpoint this can be a huge improvement as the application developer has a complete understanding of the application and can make better educated decisions on when or where to execute certain functionality. A potential drawback of this approach is the increased complexity in the user application code.

Direct3D 11 and especially OpenGL 4 both feature a very limited concurrency model. In OpenGL calls to the API can only be made from one thread at a time, and switching this thread is very costly. In Direct3D resource creation is at least free threaded but rendering is done either on a single thread or through deferred contexts which have a high overhead. In Vulkan and Direct3D 12 the API user is responsible for managing concurrency and synchronizing resource usage. To do so different synchronization primitives are available to make sure resources currently in use by the GPU are not changed by the CPU and vice versa. Command submission itself has been reworked too and now supports recording command buffers on any

[7] https://msdn.microsoft.com/en-us/library/windows/desktop/dn903821(v=vs.85).aspx

thread in both APIs. These command buffers can then be submitted from the main rendering thread to make sure their work is done in the correct sequence.

GPU memory management is in the hands of the application developer as well. In the old APIs resources like textures or vertex buffers do allocate the needed memory when created. In the new API designs the application developer allocates the memory upfront instead and freely creates resources using this memory. In addition the resource types are also simplified by only distinguishing between images, which use the texturing hardware, and buffers.

The new graphics APIs are an important step in the direction of higher performance rendering, but they require a massive amount of code to be changed to efficiently use the new functionality.

3 Related Work

3.1 nVidia Bindless OpenGL Extensions

In 2009 nVidia presented a set of extensions for OpenGL in [4] to allow access of resources without the need to bind them to the pipeline first.

Part of these extensions was GL_NV_shader_buffer_load [5]. This extension allows shaders to use buffer objects directly without binding. The main entry points exposed are *GetBufferParameterui64vNV()* and *MakeBufferResident()*. The function GetBufferParameterui64vNV() is used to get a 64 bit handle to an existing buffer object. This handle can then be made resident via MakeBufferResident(). After a handle is made resident it is accessible from the shader. The buffer handle is then transferred to the shader like a normal uniform value and is casted to the appropriate buffer object type. From there on the bindless buffer object can be used like any normal buffer object.

Later nVidia introduced a second extension called GL_NV_bindless_texture [6] that was ever since ratified by the ARB under the name GL_ARB_bindless_texture [7]. This extension is almost identical in function to the shader buffer load extension but is targeted at textures. Instead of buffer objects it operates on sampler objects and allows the bindless sampling of textures based on sampler object handles. The extension provides similar functions to GetBufferParameterui64vNV() called *GetTextureHandleARB()* and *GetTextureSamplerHandleARB()*. Both create a handle to a texture object. The second one directly associates a texture sampler object to the texture. Handles are made resident using the *MakeTextureHandleResidentARB()* function and are removed from residency using *MakeTextureHandleNonResidentARB()*. After a handle has been acquired it is not allowed to change the texture object or the sampler object associated with it. Unused handles are automatically claimed by the implementation as soon as the associated texture object or sampler object is deleted.

This set of extensions allows to considerably reduce the state switching overhead per draw call. Unfortunately the support for the extensions is still not very widespread. They are only supported by nVidia hardware based on the Kepler architecture or newer and AMD hardware with GCN architecture. At the time of writing the support by Intel GPUs was missing as well as general support on MacOS X and Linux Mesa drivers [11].

3.2 Introducing the Programmable Vertex Pulling Rendering Pipeline

Riccio and Lilley describe a way to increase draw call performance on OpenGL 4+ in their GPU Pro 4 article *Introducing the Programmable Vertex Pulling Rendering Pipeline*[8].

Their first goal was the elimination of pipeline state changes by reducing the differences in bound resources per draw call. The major contributors to these changes are textures, vertex data and shaders. Whenever one of these resources has to be changed they incur a CPU performance overhead caused by validation in the API. Additionally they also reduce the performance on the GPU side as such state changes are not free and can in some cases even stall the pipeline.

To reduce the texture resource switching they proposed to use texture arrays or, on supported hardware, sparse texture arrays. This allows shaders to have access to all loaded textures in a scene at a single resource binding point.

To eliminate the need for pipeline changes when rendering different geometry, they propose to replace the usage of Vertex Array Objects (VAOs) and automatically generated vertex fetching by a programmable vertex fetching pipeline. Instead of being stored in VAOs, the vertex attributes are kept in buffer objects and the vertex shader is responsible for fetching the necessary data. This allows them to have a small number of geometry buffers containing all required scene objects, which in turn reduces the necessary draw calls substantially.

Their approach to culling objects outside the camera view is similar to the one described later in this thesis and employed by the *breach* engine. They use a compute shader to decide the visibility of an object and generate the appropriate draw call arguments based on the outcome of the visibility test. This argument generation is where their approach diverges from this thesis as the necessary multi-draw-indirect functionality does not exist in Direct3D. Instead we directly generate a new index buffer for rendering.

The actual draw submission is done using the OpenGL 4.3 function glMultiDrawElementsIndirect(). The function uses an array of draw arguments for instanced rendering that is allocated in GPU memory. This argument array is used to draw a number of instances using multiple offsets into a single element buffer.

3.3 Improving Performance by Reducing Calls to the Driver

Hillaire describes in Chapter 25 of [15] different ways to improve application performance by reducing calls to the OpenGL API.

Part one of the chapter concentrates on the reduction of OpenGL state changes by filtering out redundant changes. As described in Section 2 the rendering pipeline contains a set of state variables that can be changed to alter the processing done in the pipeline. In OpenGL these state variables can be set by calling specific functions like glEnable, glDisable, glDepthFunc, etc. The cost of such a function call depends fully on the driver implementation, therefore state change calls should only be made when necessary. Hillaire recommends to keep track of the active OpenGL state on the CPU side. With this information it is simple to first check if a state is already set to the desired value, and if so, not to call the corresponding OpenGL function.

The next part describes three different ways to combine multiple draw calls into a single one. This method is also called batching as it creates a batch of objects that are processed as a single unit.

combine
: Multiple objects are packed into the same vertex and index buffers. Although this method enables us to draw multiple objects with one draw call, it removes the ability to move or cull the objects independent of each other.

combine+element
: This is an extension of the first method. Instead of packing both vertex and index buffers, only vertex buffers are packed. A separate dynamic index buffer is created and filled with the vertex indices necessary to render multiple objects in one call. This allows us to cull individual objects before rendering the batch.

dynamic
: The final method will pre-allocate dynamic vertex and index buffers that are dynamically filled on the CPU and then rendered in one step. Depending on the object count in the scene this way will consume a lot of memory, but it can in some cases improve performance.

The second batching method described by Hillaire is similar to the algorithm proposed in this thesis with the difference that it is fully done on the CPU while our algorithm was designed to efficiently work on the GPU.

The last part of the chapter explains the usage of hardware instancing to improve

draw performance. By using special draw functions (e.g. glDrawElementsInstanced) the GPU driver can be instructed to draw a specific objects multiple times. Additional information per object can by provided to the shader in an array that is indexed by gl_instanceID. Although the use of instancing allows us to draw hundreds or thousands of objects, it is more limiting than the previously described general batching strategies as it is designed to render the same geometry multiple times.

4 Libraries and Utilities

It would not have been feasible to develop all necessary parts to create a fully functional rendering engine from scratch in a reasonable time frame. Therefore a small set of already available libraries have been used as they were, or have been adapted to work in cooperation with the developed software system.

4.1 SDL

The Simple DirectMedia Layer[17] or SDL is a library developed by Sam Lantinga et al. to enable easier multi platform creation of games and media applications. It allows developers to gain low level access to audio and video output devices as well as input devices like keyboard, mouse or game controllers. The library is available for Windows, Mac OS X and Linux as well as mobile operating systems. The parts of the library used in *breach* include window creation, input handling and file IO.

More on window creation, input handling and file IO can be found in Section 5.

4.2 glm

OpenGL Mathematics[21] (glm) is used to provide vector and matrix math functions to the engine. It is a header only library developed by Christophe Riccio that is included into the application source code and compiled directly with the application. No further libraries or other dependencies are needed which makes the library very simple to integrate. Additionally it is easily portable to multiple operating systems. All core types as well as functions and operators are designed to directly match the facilities provided by the OpenGL shading language GLSL. In addition to the functions presented in the GLSL specification glm provides multiple extensions like quaternions or utility functions to generate different types of transformation matrices.

Although it is designed to be directly used with OpenGL and GLSL, nothing prevents the usage with Direct3D and HLSL.

4.3 ImGui

To enable the engine to display debug information and to allow the user to change the available settings a simple GUI system is required. Therefore a very bare bone 2D renderer was developed from scratch to draw simple two dimensional shapes and bitmap based text. The text rendering was based on the stb_truetype library[3]. Although this was enough to display simple information like frame timings, soon the need for user interaction arose and a more powerful alternative was incorporated.

The ImGui[9] library is a very fast immediate GUI system written by Omar Cornut et al. It only consists of four files that have to be added and some simple interface functions that are easy to implement. The immediate mode UI paradigm allows for very few code that has to be added to achieve user interaction. It allows the engine do display information in multiple ways via text and graphs and also supports different input widgets like buttons, spinners and text fields. Listing 1 contains a short example of C++ code showing the way ImGui is used in *breach*. Figure 3 shows the debugging user interface displaying frame timings and render settings.

Figure 3: The user interface in *breach* is provided by ImGui.

Listing 1: Example GUI code using ImGui.

```
// Collapsible region in the active window.
if (ImGui::CollapsingHeader("Rendering")) {
  // Buttons return if they are clicked.
  if (ImGui::Button("Reset Renderer")) {
    breach::rendering::Reset();
  }

  // Labels are text displays with a describing label.
  ImGui::Value("Lights: ", breach::rendering::_state.debug.DrawnLights);
  ImGui::Value("Draw Queues: ", breach::rendering::_state.debug.UsedQueues);

  // Checkboxes directly change the value of the supplied pointers.
  ImGui::Checkbox("Wireframe", &breach::rendering::debugOptions.DrawWireFrame);
  ImGui::Checkbox("AABB", &breach::rendering::debugOptions.DrawAABB);
  ImGui::Checkbox("Light Grid", &breach::rendering::debugOptions.DrawLightGrid);

  // Layout info is supplied inline.
  ImGui::Columns(2);

  // This will display all loaded mesh resources with their source path.
  for (it_type iterator = _state.geometryLookup->_lookup->begin();
       iterator != _state.geometryLookup->_lookup->end();
       iterator++)
  {
    std::string* path = breach::filesystem::GetPath(iterator->first);
    if (path) {
      ImGui::Text("%I64u (%s)", iterator->first, path->c_str());
    } else {
      ImGui::Text("%I64u (not available)", iterator->first);
    }
    ImGui::NextColumn();
    ImGui::Text("%i", lookUp->GetUsage(iterator->second));
    ImGui::NextColumn();
    ImGui::Separator();
  }
}
```

4.4 STB

The stb libraries are a collection of portable header-only libraries[3] developed by Sean T. Barrett. They consist of one header file per library that can be included on demand and ranges in functionality from reading textures, decoding sound files and rasterizing font files to text editor functions and a c-lexer. The *breach* engine makes use of the texture loader stb_image as it allows loading and decoding of file formats like JPG, PNG, TGA, PSD and more. The library is used by the texture conditioner in *breach* to load raw bitmap assets and transform them into run-time texture data.

As mentioned before stb_truetype was used before switching to ImGui to rasterize text.

4.5 Assimp

The Open Asset Importer Library[12] (assimp) written by Alexander Gessler et al. is a multi platform asset importer library. It is able to load different 3D object file formats like Filmbox (.fbx), Wavefront Object (.obj) or Collada (.dae) into a single unified representation. This representation is used by the geometry conditioner in *breach* to extract the index and vertex buffers from 3D assets and convert them into the optimized run time representation used by the renderer.

The assimp library is distributed as dynamic and static link library and comes with both a C and C++ interface. Listing 11 at the end of this thesis contains the implementation of the geometry conditioner using the assimp library.

4.6 RapidJSON

The *breach* engine supports loading meta data that is supplied in JSON formatted files. This meta data is mainly used to define assets having no other common representation. For both geometry and textures, widely used standardized file formats exist, therefore materials and scene data are the primary assets supplied in JSON. The RapidJSON [23] library by Milo Yip was integrated to allow parsing of these JSON files. Like all assets the JSON files are parsed by a resource conditioner and transformed into a binary run time format that is faster to load. See section 6 for more information on resource conditioning.

4.7 DirectXTex

The DirectXTex [18] library is a utility library provided by Microsoft under the MS-PL license that allows reading and writing of DDS file formats. Additionally it supports conversion between different texture formats. It is used in the texture conditioning process to generate optimized compressed textures from the raw bitmaps loaded by stb_image.

5 Engine Architecture

This Section contains detailed information about the structure and architecture of the rendering engine *breach*.

The engine was developed in C++, but only very few object oriented features where used. There is almost no inheritance and most structures are only plain old data structures (PODs) without member functions. This is mostly due to personal preference of the author, as subjectively deep inheritance and overuse of object oriented patterns tend to complicate the code and make it more difficult to understand. The lookupTable class is the only class making use of both member functions and templates, mainly because it seemed easier at the time of writing.

Compilation of the engine is done as a unity build. Instead of compiling all cpp files separately, which can invoke the compiler hundreds of times per compilation, all code is included into one source file (main.cpp) which is then compiled. This allows the engine to be compiled from scratch in release mode in under 6 seconds (4 seconds for debug builds). The same source code compiled separately needed more than 35 seconds to compile.[8] Figure 4 shows the include structure of the engine.

C++ namespaces are employed to structure the engine modules semantically. The following sub sections describe all of the major namespaces.

5.1 breach

The breach namespace contains functions to initialize and to unload the engine. All other parts are initialized from this starting point by calling the initialize function of the respective namespace. In addition to initializing all other modules the breach initializer also creates the application window using SDL. To be able to use GPU acceleration on Windows in connection with SDL created windows the SDL_WINDOW_OPENGL flag has to be provided to the SDL_CreateWindow function. The platform agnostic SDL window is then handed to the target platforms graphics initializer to create the correct rendering context.

The breach namespace also contains basic types used in the whole engine like fixed width integer types and the hash_string class that is used for resource management.

[8] Compiled on an Intel Core i5-3570K with 16316 MB RAM.

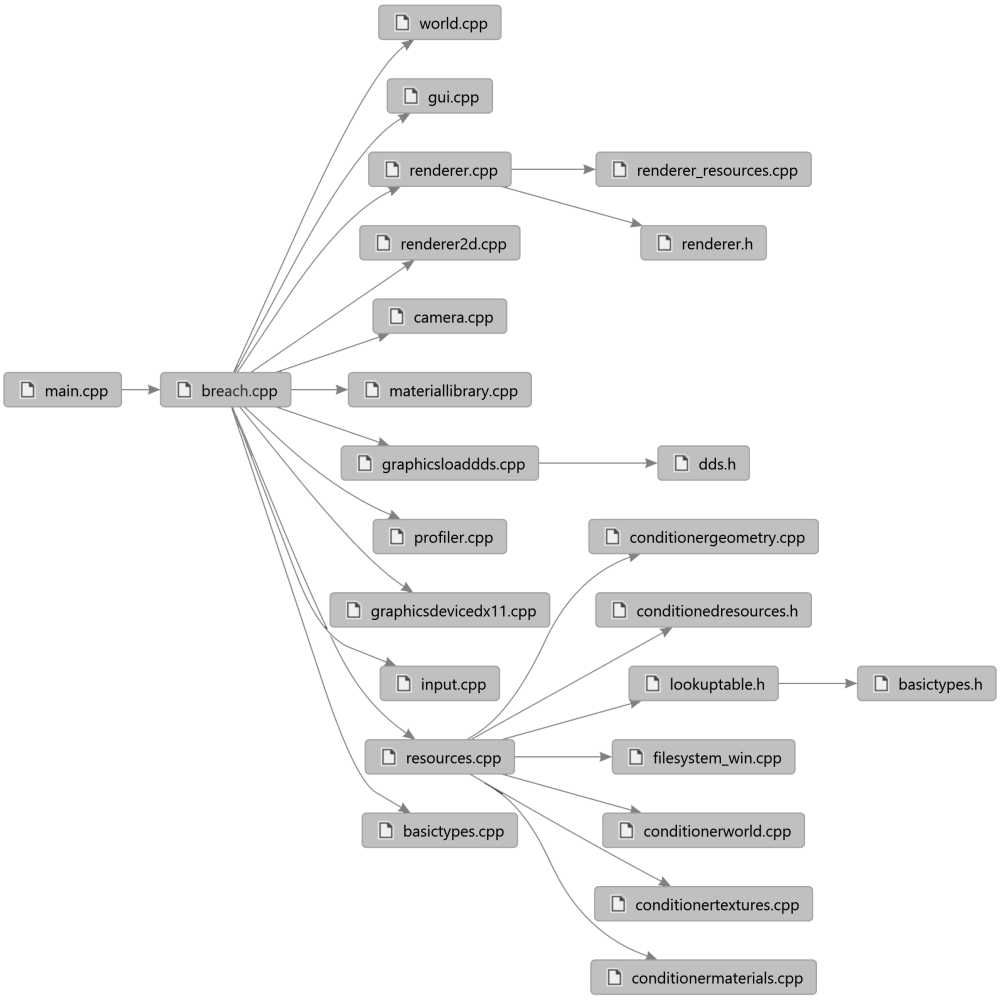

Figure 4: Include dependencies of the breach engine.

5.2 graphics

The graphics namespace contains wrapper structures and functions that hide the underlying graphics API from the rest of the engine. The current wrapper is written against the Direct3D 11 API, but can potentially be replaced with a wrapper written against any other graphics API. The interface of the abstraction layer allows for the creation of hardware resources like textures, vertex buffers, index buffers, frame buffers and shaders. It also provides functions to change state variables like raster state or fill state and is used to issue draw calls and compute shader dispatches. For performance measurement the API provides the ability to create hardware performance queries. These queries can be used to time actions on the GPU and are further elaborated on in the next section.

5.3 profiling

The profiling namespace implements timing functions for both CPU and GPU tasks. To time a certain function of part of code it can be wrapped between StartCPUQuery() and StopCPUQuery() statements. The time between the two calls is recorded using the multi-platform SDL performance counter. For measuring GPU performance the functions StartGPUQuery() and StopGPUQuery() can be used. They will insert timestamp events into the command buffer of the GPU using the graphics module. As the GPU can potentially be multiple frames behind the CPU in execution the result of the GPU query can not be directly returned to the CPU without stalling the CPU for multiple frames. Therefore the profiler will remember all queries done in a certain frame and will, after a safe amount of frames, gather the results for the hardware queries from the GPU.

The results of the performance queries can be displayed on screen using ImGui or be stored in a JSON file for later evaluation. The built in profiler was the main source of performance measurements used in the development and optimization of *breach* apart from CPU and GPU instrumentation software like GPUPerfStudio or CodeXL.

5.4 input

The input namespace provides functions to retrieve mouse movement and keyboard interaction. The main application contains a message loop that consumes events provided by the SDL. Whenever a new SDL_Event is encountered by the main loop the input::Update() function is called which updates the engine internal state with the event content. All other engine modules can then directly access the user input from the input module. Listing 2 contains the code for the whole input module.

Listing 2: SDL input wrapper.

```
namespace breach
{
  namespace input
  {
    breach_global_variable bool _keysPressed[283];
    breach_global_variable bool _mouseButtonPressed[8];
    breach_global_variable float _mouseMovementX;
    breach_global_variable float _mouseMovementY;

    breach_external void NewFrame() {
      _mouseMovementX = 0;
      _mouseMovementY = 0;
    }

    breach_external void Update(SDL_Event event) {
      if (event.type == SDL_KEYDOWN) {
        _keysPressed[event.key.keysym.scancode] = true;
      }
      else if (event.type == SDL_KEYUP) {
        _keysPressed[event.key.keysym.scancode] = false;
      }
      else if (event.type == SDL_MOUSEBUTTONDOWN) {
        _mouseButtonPressed[event.button.button] = true;
      }
      else if (event.type == SDL_MOUSEBUTTONUP) {
        _mouseButtonPressed[event.button.button] = false;
      }
      else if (event.type == SDL_MOUSEMOTION) {
        _mouseMovementX = (float)event.motion.xrel;
        _mouseMovementY = (float)event.motion.yrel;
      }
    }

    breach_external bool IsKeyPressed(SDL_Scancode key) {
      return _keysPressed[key];
    }

    breach_external bool IsMouseButtonPressed(uint8 button) {
      return _mouseButtonPressed[button];
    }

    breach_external float GetMouseMovementX() {
      return _mouseMovementX;
    }

    breach_external float GetMouseMovementY() {
      return _mouseMovementY;
    }
  }
}
```

5.5 filesystem

The filesystem namespace contains all functionality that is needed to access files. The resource module of the engine uses hashed paths (a string hashed into a 64 bit unsigned integer) to identify resources. When initializing the engine an asset folder has to be specified. All paths of the files inside this folder are hashed and added to a look up table. Whenever another module needs access to a file the filesystem can be queried for the hashed path and returns a SDL_RWops object. This can then be used in conjunction with standard SDL read and write functions to access the file. Additionally the filesystem allows to directly read the whole file content and to register a callback whenever a file changed. This can be used to implement hot reloading of resources, but is currently not in use.

5.6 gui

The gui namespace provides the necessary implementation of the ImGui interface. It contains code to initialize ImGui and its resources, to transfer the user input from the input module to the ImGui library and to access the system clipboard for copy and paste operations. It also implements the rendering interface of ImGui. The ImGui rendering interface consists of a single function pointer that has to be set to a function that receives an array of command lists. Each command in the list either draws a list of triangles or pushes/pops a clipping rectangle onto a stack. ImGui comes with multiple examples on how to implement the rendering function using different graphics APIs. The rendering function supplied by *breach* is a modified version of the OpenGL 3 example implementation that uses the *breach* graphics module instead.

5.7 resources

The resources module is a collection of functions to convert assets like meshes, textures or materials from a general purpose format into an optimized run-time format used by the engine. This process of conditioning is described in more detail in the Section 6.

5.8 world

The world module manages scene creation and rendering. A scene in *breach* is defined by the WorldState structure. Listing 3 shows the implementation of this structure. The WorldState contains a table to map from hashed entity names to indices in the EntityInfo array. This array contains the indices to the actual components making up an entity. In the case of *breach* only two components WorldStaticMeshData and WorldLightData are available. Additional components can easily be added to augment entity behaviour. Both the local and global transform for each entity are stored in arrays accessed by the same index as the EntityInfo array.

A world is drawn by submitting the data of all active WorldLightData and WorldStaticMeshData components to the rendering module using the SetLightDataXXX() and Draw() functions.

Listing 3: Data definition of the *breach* world state.

```
struct WorldLightData
{
  size_t entityIndex;
  float fluxCandela;
  glm::vec3 colorFilter;
  float areaSize;
  float maxRadius;
  rendering::BREACH_LIGHT_TYPE type;
  uint32 shadowProjectors[6];
};

struct WorldStaticMeshData
{
  size_t entityIndex;
  uint32 materialIndex;
  uint32 geometryIndex;
};

struct WorldEntityInfo
{
  size_t staticMeshIndex;
  size_t lightIndex;
};

struct WorldState
{
  std::vector<HashString> EntityNames;
  std::unordered_map<uint64, size_t> EntityLookup;
  std::vector<WorldEntityInfo> EntityInfo;
  std::vector<glm::mat4> LocalTransforms;
  std::vector<glm::mat4> WorldTransforms;
  std::vector<WorldStaticMeshData> StaticMeshData;
  std::vector<WorldLightData> LightData;
};
```

5.9 rendering

The rendering module is the biggest part of the *breach* engine. It contains all renderer specific code for managing resources and drawing them. In addition to the code implementing the rendering algorithm described in Section 8 it also contains all necessary functionality for a second renderer used to compare the algorithms performance.

A large portion of the rendering module consists of code to manage graphics resources. The material library, which is used to efficiently load textures and materials when they are needed, is located here and employs reference counting to free loaded resources if they are not used anymore. Geometry is managed in a similar fashion directly by each renderer as they have different requirements concerning the loaded assets. More details on the resource management of the renderer can be found in Section 7.

All rendering is done in the SubmitFrame() function of this module. Based on the currently active renderer a different set of functions is executed. As mentioned in the description of the world module, the renderer provides an interface to register lights and objects to be drawn. The SetLightDataOmni() and SetLightDataDirectional() set of functions is independent of the active renderer, but the Draw() function will execute different code depending on the algorithm in use. More information on the actual implementation can be found in Section 8.

5.10 rendering2d

The rendering2d module uses the functionality of the graphics and rendering modules to draw simple shapes like lines, spheres or bounding boxes. These can be used for debugging purposes. Before ImGui was introduced in *breach*, the rendering2d namespace contained code to draw text and textures to the screen. Since then this functionality was removed.

6 Resource Conditioning

A *resource* or *asset* in the sense of this thesis is a piece of data that has a distinct application in the engine. Resources include geometric information (meshes), textures, materials and worlds. To convert the raw data that is created by the user or exported from content creation tools into a format that is usable by the engine a process called conditioning is employed. The data is loaded by a conditioning function and transformed into a binary format. This ensures that the core engine is kept independent of the input formats. Therefore additional file formats for assets can easily be added without changing the core.

All resources in the engine are indexed using a hashed path. The user supplies the relative path to the desired resource, this path is then hashed using a simple hashing function to generate a 64 bit unsigned integer. On start up or when commanded by the user, the asset directory is scanned for files and all found file paths are hashed and stored in a look-up table.

Whenever a resource (mesh, texture, material, etc.) is required by the user the resource loading system will first look for the hash in an on disc cache. This cache stores the already conditioned binary resources that are ready to be used by the engine. The cache file names are based on the original hash, a time stamp, the conditioner version and a user supplied usage parameter. The usage parameter decides which conditioner should be used when a resource is loaded. It is possible that the same resource is loaded using different conditioners, see Section 6.4 for more information on this. Whenever the cache is missed the system will trigger the conditioning of the resource. The conditioner will then load the raw resource from the disc using the look-up table and call the conditioning function according to the usage parameter.

After the resource has been successfully conditioned it is stored in the cache in a binary format and the engine will continue to load the conditioned resource.

6.1 Materials

Materials are described using a simple JSON formatted file. Currently the engine supports only one type of material with the following information stored: base color, roughness, specular intensity, anisotropy and metalness. In addition to these numeric values the paths to the five texture types (albedo, alpha, normal, metalness, roughness) are stored in the JSON file. Listing 4 shows an example material file. When conditioned, this file is converted to a binary representation that stores the same information numerical wise. The paths however are not stored as plain strings, but are hashed using the same algorithm that the resource loading code uses. This allows the engine to directly find the resource without storing a real string. Listing 5 shows the C++ structure used to store the material runtime information.

Listing 4: Material JSON source file

```
{
  "albedoPath" : "textures/asteroid/stone_10_diffuse.tga",
  "alphaPath" : "",
  "normalPath" : "textures/asteroid/stone_10_normal.tga",
  "metalnessPath" : "",
  "roughnessPath" : "",
  "baseColor" : {
    "r" : 1.0,
    "g" : 1.0,
    "b" : 1.0,
    "a" : 1.0
  },
  "metalness" : 0.0,
  "roughness" : 0.8,
  "specularIntensity" : 0.5,
  "anisotropy" : 0.0
}
```

Listing 5: Material runtime information

```
struct ConditionedMaterial
{
    HashString AlbedoTexture = HashString();
    HashString AlphaTexture = HashString();
    HashString NormalTexture = HashString();
    HashString MetalnessTexture = HashString();
    HashString RoughnessTexture = HashString();
    glm::vec4 BaseColor = glm::vec4(1.0f, 1.0f, 1.0f, 1.0f);
    float Roughness = 0.5f;
    float SpecularIntensity = 1.0f;
    float Anisotropy = 0.0f;
    float Metalness = 0.0f;
};
```

6.2 Geometry

The engine supports the conditioning of all geometry file formats supported by the assimp library[9]. All test scenes used in this thesis consist of meshes exported from 3D Studio Max using the Filmbox (*.fbx) file format.

When a 3D object is conditioned, the assimp library is invoked to load the mesh into a documented intermediary format. From that point on the necessary information is extracted and stored in the binary format used by the engine. The binary format can be seen in Listing 6, it is optimized for fast loading and stores the vertex information in a compressed form that is exptected by the engine.

Vertex compression is done by packing two 16 bit half-precision floating point values into one unsigned 32 bit integer where applicable. In the case of the reference renderer the unpacking is done automatically by the graphics driver when a vertex is fetched. The *breach* renderer unpacks the values manually in the shader. More information on this subject can be found in Section 8. The code for unpacking is shown in Listing 9.

Listing 6: Geometry runtime information

```
struct ConditionedGeometryDataVertex
{
    glm::vec3 position;
    uint32 packedTexCoord;
    uint32 packedNormalXY;
    uint32 packedNormalZmaterialIndex;
};

struct ConditionedGeometryDataHeader
{
    uint32 vertexCount;
    uint32 indexCount;
    uint32 materialCount;
};

struct ConditionedGeometryData
{
    ConditionedGeometryDataHeader header;
    ConditionedGeometryDataVertex* vertices;
    uint32* indices;
};
```

[9] see `http://assimp.sourceforge.net/main_features_formats.html` for full format listing

6.3 World Data

The world data describes the objects and resources that have to be loaded to render a scene. This world data is stored in JSON and is conditioned like the other resources. Listing 7 shows the world data file of a simple scene. The format currently contains two arrays. One for entities, which are the objects to be rendered, and one for the lights. The entities store a name for referencing them in code or other entities, a transform consisting of Cartesian coordinates, non-uniform scale and a quaternion for rotation. In addition it stores the paths to the geometry and material files. Lights contain the same name and transform information but specify light parameters instead of geometry and material information. The light information specifies if the light casts shadows, the maximum radius, the flux of the light in Candela and a filter color. When conditioned all paths are converted to hashed strings and the transform information is converted into a world matrix. Listing 8 shows the structure of the runtime binary format.

Listing 7: World JSON file

```
{
  "entities" : [
    {
      "name" : "teapot",
      "transform" : {
        "position" : {"x":0.0,"y":0.0,"z":0.0},
        "scale" : {"x":0.01,"y":0.01,"z":0.01},
        "rotation" : {"x":0.0,"y":0.0,"z":0.0,"w":-1.0}
      },
      "staticMesh" : {
        "meshPath" : "meshes/Teapot.fbx",
        "materialPath" : "materials/Teapot.material"
      }
    }
  ],
  "lights" : [
    {
      "name" : "PhotometricLight001",
      "transform" : {
        "position" : {"x":-5.49059,"y":14.9155,"z":8.68616},
        "scale" : {"x":1.0,"y":1.0,"z":1.0},
        "rotation" : {"x":0.0,"y":0.0,"z":0.0,"w":-1.0}
      },
      "lightInfo" : {
        "shadows" : true,
        "maxRadius" : 40.0,
        "fluxCandela" : 200.0,
        "filter" : {"r":255.0,"g":255.0,"b":255.0}
      }
    }
  ]
}
```

Listing 8: World runtime information

```
struct ConditionedWorldDataEntity
{
    HashString name;
    HashString meshPath;
    HashString materialPath;
    glm::mat4 worldMatrix;
};
struct ConditionedWorldDataLight
{
    HashString name;
    glm::mat4 worldMatrix;
    bool shadows;
    float maxRadius;
    glm::vec3 filter;
    float fluxCandela;
};
struct ConditionedWorldData
{
    uint32 entityCount;
    uint32 lightCount;
    ConditionedWorldDataEntity *entities;
    ConditionedWorldDataLight *lights;
};
```

6.4 Textures

Textures are handled a bit differently from the other conditioned resources. When loading a resource in the engine the user always specifies a usage parameter. This parameter decides what type of conditioner to use for the resource. This system is used to load the same texture in different formats. There are currently five different types of textures employed by the engine: diffuse, normal, alpha, roughness and metalness textures. The diffuse texture describes the color of the surface. The normal texture defines the surface normals and the alpha texture stores the transparency. Metalness and roughness are used to vary the metalness and roughness parameters of a surface. By supplying a usage parameter when requesting a resource, it is possible to load the same raw texture to be used as diffuse texture and as alpha mask.

Figure 5 shows the different texture types used to create a rock material. The diffuse map describes the overall color of the rock while the normal map describes fine surface details too small to model with geometry. The roughness map describes how light is reflected, in this example the areas extruding from the surface are smoother (lighter) than the crevices between (darker). The metalness map in this example is black because the rock does not consist of metal parts. The alpha is fully white because there are no transparencies.

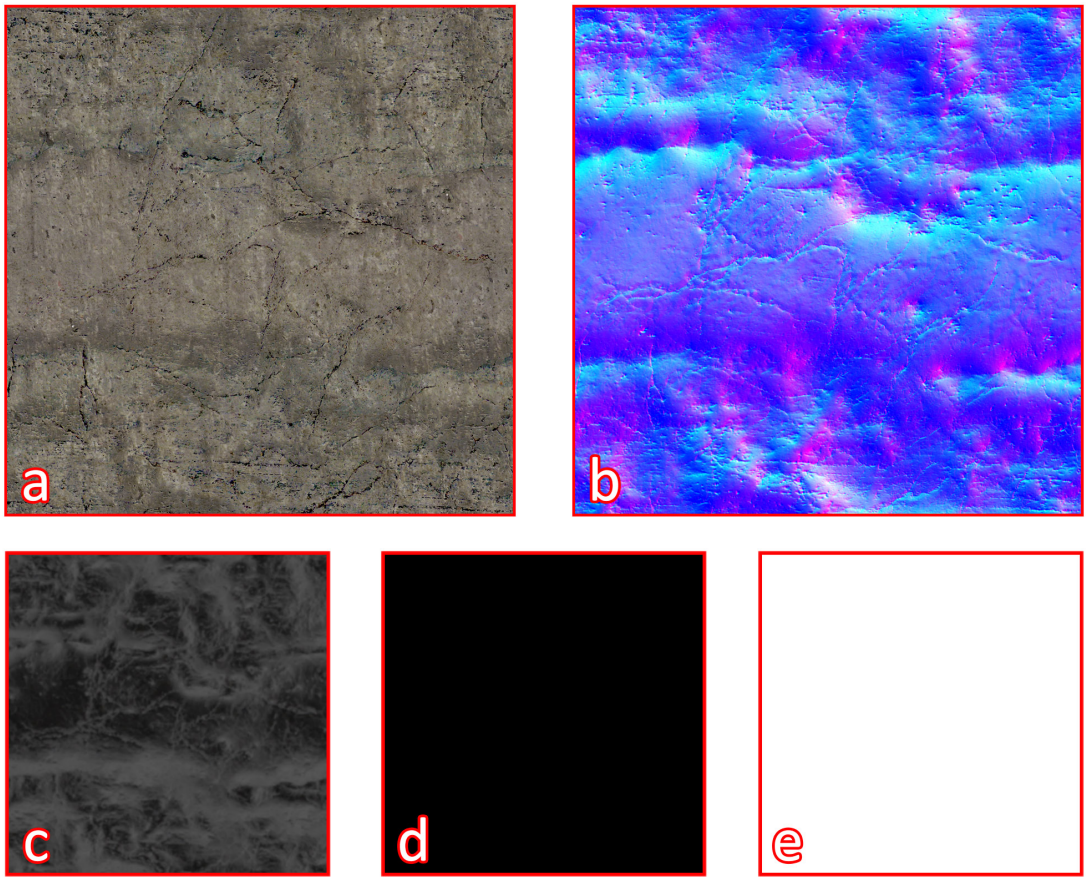

Figure 5: Texture types in *breach*: a. diffuse, b. normal, c. roughness, d. metalness, e. alpha

The conditioner stores the textures in the DirectDraw Surface format (*.dds)[10]. This format allows to store compressed textures that are directly supported by the graphics hardware. The different texture types are compressed using different format options. Diffuse textures are stored as BC1 in sRGB mode which allows for a compressed size of 0.5 bytes per pixel. Normal textures use BC5 compression which stores two channels (the third is reconstructed in the shader) as 1 byte per pixel. Alpha, metalness and roughness have only one channel each, allowing the use of high quality BC4 compression. This format is designed to compress single channel images to 0.5 bytes per pixel.

[10]see `https://msdn.microsoft.com/en-us/library/windows/desktop/hh308955(v=vs.85).aspx` for more information on the DDS format and Block Compression

7 Resource Management

This section describes the constraints that the proposed rendering system imposes on the resource management system and the design decisions made in the *breach* implementation.

7.1 Meshes

One way to load meshes into a game engine is to create one index and vertex buffer for each mesh. These buffers are then used for drawing. To be able to access all geometric data from a compute shader it is necessary to use single index and vertex buffers for all meshes. When loading a mesh, its vertex and index data is simply appended to the data in the global vertex and index buffer. When removing a mesh there are multiple options:

Compact on removal

The resulting hole is removed by moving all following meshes up to fill the hole. The cost of this technique depends on the number of meshes loaded and the position of the removed mesh. This method potentially has to move a lot of data, but can be improved by only moving a fixed amount of meshes per frame to reduce potential frame time spiking. Because of its simple implementation it was used in *breach*.

Compact on fail

Each removal will cause a hole that is accepted as long as there is enough memory in the back of the buffer. When the creation of a new mesh fails the buffer is compacted as described before.

Hole filling

On each removal an entry is added to a free list describing the position and size of the gap in the buffer. When a new mesh is created it tries to fill the smallest possible gap. If there exists neither a gap big enough nor enough memory at the end of the buffer some gaps need to be combined.

Fixed size chunks

The buffer is grouped in fixed size chunks. The general algorithm is the same as with the *hole filling* method, but it is somewhat easier to manage as the gaps can not be of arbitrary size.

7.2 Textures

All textures have to be available to the pixel shader to be able to use multiple textures in a single draw call. One possibility to achieve this is to use bindless resources[4]. In this case no textures are directly bound to the pipeline, instead they can be accessed through a resource handle that is stored in the material.

Another way to enable quasi-bindless rendering, if bindless resources are not exposed by the target API or hardware, is to use a texture atlas or texture array. A texture atlas is a big texture that contains multiple smaller textures layed out in a regular or irregular fashion. No special hardware support is needed for this approach, but the texture count is constrained by the maximum size of a single texture, and the uv-coordinates used to address a texture in the atlas have to be adjusted based on the position of the texture inside the atlas. An additional drawback are texture sampling artifacts that can occur when sampling exactly at the border of a texture. Based on the wrap mode specified for the texture the filter function needs to fetch adjacent texels for interpolation. If the border of the physical texture (the atlas) and the logical texture (the smaller texture on the atlas) do not coincide it will instead sample from the neighbouring texture. There are at least two ways to counter this behaviour.

The first is to implement texture filtering directly in the shader. The shader knows the uv-coordinates of the border and can fetch the correct pixels. Depending on the hardware this can impact performance in addition to the added complexity of the shader.

The second way is to add a border of pixels around the texture on the atlas. The addition of the wrapped pixels on each side of the texture will allow texture sampling to work normally. The desired quality of texture filtering will dictate the width of the border. For standard bi-linear filtering a one pixel border is sufficient while anisotropic filtering needs additional pixels based on the anisotropy factor. In addition to added complexity at texture load or bake time, it will also waste some texture memory.

An alternative to texture atlases is the use of texture arrays if the hardware supports them. A texture array is a collection of multiple textures with the same texture format and size. Each texture in the array is equivalent to a normal texture for the hardware but the array of all textures can be bound as a single entity to a single binding slot of the graphics API. This allows a shader program to use a multitude of different textures that only occupy a single binding slot. The count of textures in

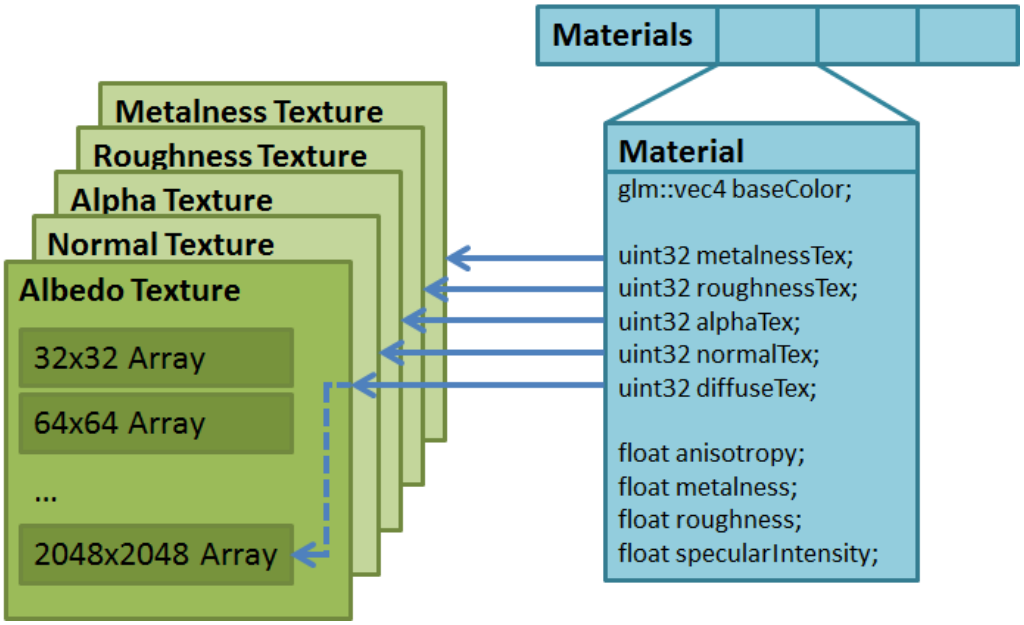

Figure 6: Material library

this case is restricted by the maximum texture array size of the underlying hardware. If supported by the hardware, sparse textures can be utilized to create a huge array of textures that don't need to be fully backed by memory. In this case the array is created with thousands of slots for textures, but only textures that are really needed are uploaded to video memory.

To support D3D11 and older video cards the texture array approach is used by *breach*. At start-up an array of texture arrays of different sizes is created and then filled with textures when they are loaded. Each texture handle in the engine gets a texture index that is composed of an index to the texture array of the specific size and the index of the slice inside the array. The pixel shader then uses this handle to sample from the correct texture array slice. Figure 6 shows a conceptual overview of the material library containing these texture arrays.

7.3 Materials

Materials contain the surface parameters needed for rendering. The binary representation supplied by the resource conditioner is loaded and the hashed texture paths are converted into the texture handles described in the last paragraph. This is done by querying the texture library of the graphics module for the texture described by the hashed string. The texture library will then return the handle to the texture, containing the encoded indices for the array and the slice. The final material data (numeric surface parameters and texture handles) are then stored as a material array in a GPU buffer and are updated whenever a new material is loaded or unloaded. When submitting an object to the renderer, the index into this material array is stored in the corresponding draw package. The material index is extracted when drawing an object and the material is loaded from the buffer to be used for shading. An overview of the material library design is shown in Figure 6.

8 Rendering

The algorithm proposed here is a combination of the methods described in [15] and [2]. Hillaire shows in [15] a batching scheme that generates dynamic index buffers on the CPU. We expand this idea by transferring the work to the GPU. This is done by using a draw package scheme that is similar to the one presented by Baker in [2]. Although most of their work creating the package queue is done on the CPU we propose a solution to move some of the computation to the GPU. The resulting queue is then used to generate a dynamic index buffer. A single draw call can then be used to submit all objects in the queue to the GPU.

As described in Section 3, [2] features a queue that contains draw packages which defines all necessary information for drawing the objects. This information consists at least of the vertex and index buffers to use, the shader, the textures and the world transform matrix. In their algorithm they collect the data and then submit it via normal draw calls on multiple threads.

We propose to submit the entire package queue instead. This enables us to carry out additional operations on the data directly on the GPU without the need for the CPU to interfere. Figure 7 shows the high level overview of the algorithms data flow on the GPU, which is described in the following subsections.

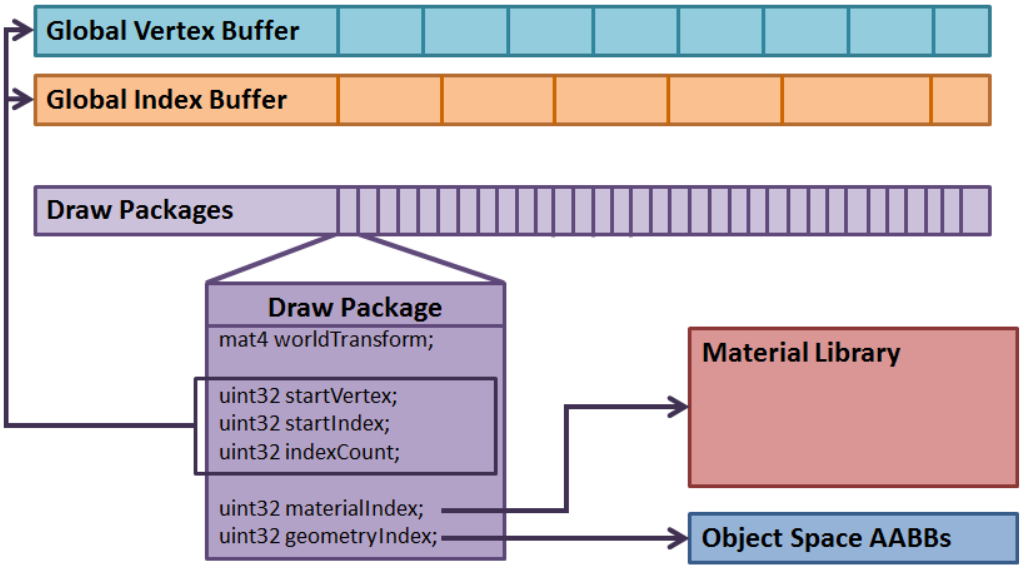

Figure 7: Data flow overview

8.1 Package generation

Whenever an object needs to be rendered, a draw package is constructed and added to a CPU sided queue. This draw package consists of all geometric information needed: the offset into the vertex buffer, the offset into the index buffer and the index count. In additional to the geometric information an index into the material library is supplied to tell the bound shader which textures and material settings to use. To define the position and orientation of the object in the world a transformation matrix needs to be stored in the package as well. If frustum culling should be performed on the GPU, an additional object index is added to the package. This index defines which pre-computed object space bounding box has to be used for frustum culling. The bounding boxes are updated analogous to the index and vertex buffers. Whenever an object is loaded, its bounding box is computed and uploaded to a GPU buffer.

After all objects are submitted to the queue, it is copied into a GPU sided buffer. From there on almost all operations are done directly on the GPU without the CPU interfering.

8.2 Draw package expansion and culling

On today's graphics hardware the drawing of triangles is done by supplying a vertex shader with vertex data defining the extent and attributes of a triangle. This can generally be done using two different approaches: indexed or not indexed. When not using indexing, the graphics card will receive a vertex count and will go through the list of vertices in the bound vertex buffer in order. Indexed rendering on the other hand supplies the vertex shader with an index into the bound vertex buffer. The advantage of indexed rendering is the potential reuse of vertices when they are shared by multiple triangles. Both approaches will automatically fetch the vertex data if a vertex buffer is bound.

Unfortunately this poses a problem for us. Our algorithm tries to only use one draw call. If we just go through all vertices that have to be rendered we have no way of identifying the draw package a single vertex belongs to. This information is needed however, to transform the individual vertices based on the transformation matrix stored in the draw package.

Therefore the first step of the GPU side pipeline is a compute shader that takes all draw packages and expands the index offsets and index counts of an object into a

single list of indices. This shader runs one work-group per draw package with each work-group consisting of 64 to 256 threads. This number can be tuned depending on the underlying hardware. The first thread in the work-group will fetch the correct draw package for its group and store it in thread shared memory. In the next step the bounding box is fetched based on the object index in the draw package and is transformed into world space. The transformed bounding box is then intersected with the view frustum. If the object is visible a thread shared visibility flag is set and an atomic counter is increased by the index count of the object. The old value of the counter before the increase is used to compute the output destination in the combined index buffer. From now on all threads in the group loop over the index range specified in the draw package and copy the index into the new combined buffer.

As mentioned before the vertex index is not enough to draw an object at the correct position in world-space. In addition to the vertices we also need the transformation matrix of the object the vertices belong to. To encode this information in the index buffer, the index into the vertex buffer is combined with the index of the draw package so that later pipeline stages are able to identify which draw package a specific vertex belongs to. The combination is done by reducing both indices to 16 bit and packing them into a 32 bit integer. This limits the object count per draw and the index count per object to 65536. The splitting of the bits can be adapted to allow for either more objects or more indices at the expense of reducing the other value.

Figure 8 shows the data flow of the compute shader responsible for the expansion. In the first step the bounding box of the object is fetched using the object index and frustum culling is performed. If the bounding box passes the test the draw package is expanded and the combined indices are written into the culled expanded index buffer. This increases an atomic counter by the index count of the draw package. The counter is later used to submit the exact amount of necessary indices to the drawing pipeline.

Listing 12 at the end of this thesis shows the full compute shader code used to expand the packet queue into a continuous index buffer.

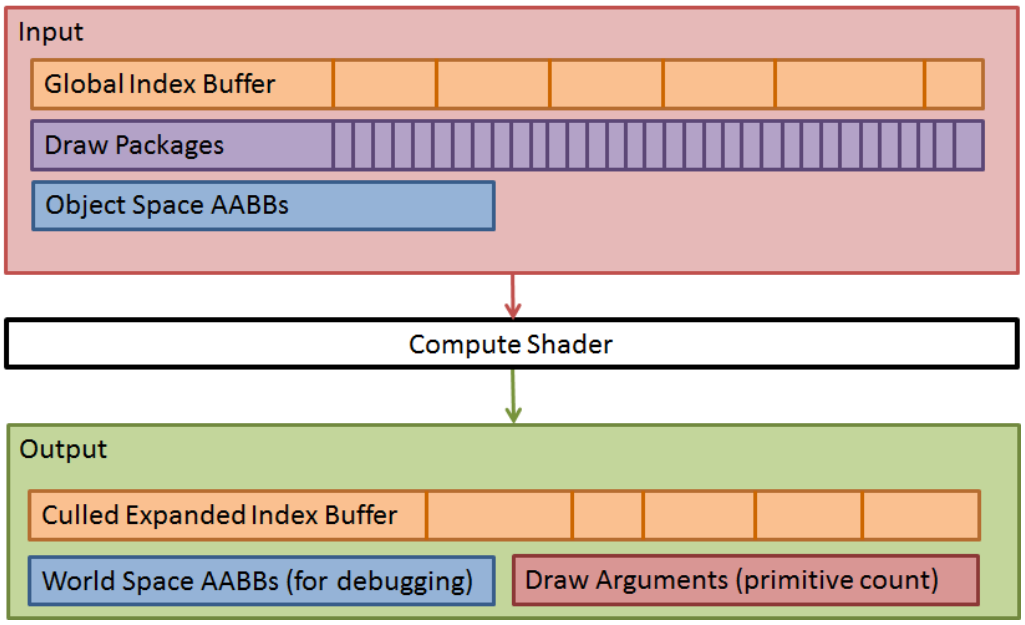

Figure 8: Expanding draw packages

8.3 Drawing

After all packages are expanded, a single draw call is issued using the index count computed in the previous stage. The counter containing the number of visible indices is stored in a so-called *draw arguments buffer*. A *draw arguments buffer* in Direct3D is a piece of GPU memory that can be used to supply the function arguments for an actual draw call. To draw all objects the combined index buffer is bound as ordinary index buffer, but there is no vertex buffer bound to the input assembler stage of the pipeline. Instead the global vertex buffer is bound as a structured buffer. When the scene is to be drawn, we use the DrawIndexedInstancedIndirect() function of the D3D11 API. It has the same functionality as the DrawIndexedInstanced() function, that is drawing multiple copies of indexed primitives, but the function parameters are directly taken from a *draw arguments buffer*. This removes the need to read-back the culled index count from the GPU.

In the vertex shader, the combined index is automatically loaded by the input assembler stage. From there on we can extract the draw package index and the actual vertex index by splitting the bit pattern into two unsigned 16 bit integers. Using the index offset and vertex offset stored in the draw package in combination with the vertex index extracted from the combined index, we can compute the final index being used to fetch the correct vertex from the global vertex buffer.

The vertex data is stored in a compressed form in the global vertex buffer by using

16 bit float values for texture coordinates and normals. When fetching the vertices directly via the input assembler stage the decompression would happen automatically. Unfortunately that is not possible in our case because of the need to decode the index of the vertex first. Therefore the decompression has to be done manually in the vertex shader. Listing 9 shows the code used to fetch a vertex and decompress it in HLSL.

Listing 9: Vertex fetching and unpacking code

```
StructuredBuffer<draw_pack> drawPacks;
StructuredBuffer<uint> indicesIn;
StructuredBuffer<packed_vertex> verticesIn;

unpacked_vertex unpack(packed_vertex v)
{
    unpacked_vertex u;
    u.pos = v.pos;

    u.tex0.x = f16tof32(v.tex0 & 0x0000ffff);
    u.tex0.y = f16tof32(v.tex0 >> 16);

    u.norm.x = f16tof32(v.normXY & 0x0000ffff);
    u.norm.y = f16tof32(v.normXY >> 16);
    u.norm.z = f16tof32(v.normZ & 0x0000ffff);

    return u;
}

PS_INPUT VS(uint index : SV_VERTEXID)
{
    uint packageIdx = (index & 0x0000FFFF);
    uint vertIdx = (index & 0xFFFF0000) >> 16;

    draw_pack pack = drawPacks[packageIdx];

    uint globalVertIdx;
    globalVertIdx = pack.startVertex + vertIdx;

    packed_vertex vpacked;
    vpacked = verticesIn[globalVertIdx];
    unpacked_vertex v = unpack(vpacked);

    float4 wPos = mul(pack.worldTransform,
                      float4(v.pos, 1));

    ...
}
```

The fetched vertex is then transformed using the transformation matrix stored in the draw package. In addition to the transformed vertex data, the material index is extracted from the draw package and submitted to the next shader stage.

In the pixel shader the material is fetched from the library using the material index submitted as vertex attribute by the vertex shader. This removes the need to fetch the draw package per pixel. The fetched material contains indices into the texture arrays. Although it is possible to define arrays of texture arrays in HLSL it is unfortunately not possible to directly access them dynamically. Therefore a switch statement was used for selection of the correct texture size. This has the effect that the sampling is conditional, which in turn prevents us from using the *Sample()* function that uses automatic texture gradients. To counteract this we generate our own texture gradients in the pixel shader and use the *SampleGrad* function instead. Listing 10 shows an abridged version of the code used for computing the texture coordinates from the index and the actual sampling of a diffuse texture.

Listing 10: Sampling of a diffuse texture

```
Texture2DArray texDiff[8] : register (t16);
float3 SampleDiffuse(SamplerState s, uint index,
                     float2 texC, float4 grad)
{
    uint sizeIndex = index >> 24;
    uint slice = index - (sizeIndex << 24);

    float3 uvw = float3(texC, slice);
    float3 d = 0;
    [forcecase] switch (sizeIndex) {
    case 0:
        d = texDiff[0].SampleGrad(s, uvw,
                        grad.xy, grad.zw).xyz;
        break;
    case 1:
        d = texDiff[1].SampleGrad(s, uvw,
                        grad.xy, grad.zw).xyz;
        break;

    ...
    case 7:
        d = texDiff[7].SampleGrad(s, uvw,
                        grad.xy, grad.zw).xyz;
        break;
    default:
        d = texDiff[0].SampleGrad(s, uvw,
                        grad.xy, grad.zw).xyz;
        break;
    }
    return d;
}
```

Finally all information needed for rendering a single pixel is present and shading can be commenced as described in Section 9.

9 Shading

Although the final output provided by the algorithm described in the previous Section can be used to shade the pixel in a wide variety of different ways, there are some constraints that have to be considered. As all objects are drawn in a single call, a basic forward shading algorithm would work but needs adaptation. Forward shading uses the CPU to compute a short list of lights influencing each object. Then for each draw call the data for all the lights in the list is uploaded to the GPU. To use this shading technique with out algorithm the light influence data could be included in the draw package. A drawback of this approach would be the considerable increase in memory needed per draw package. To mitigate this to a certain extent the renderer could keep a list of all lights of the scene in a GPU buffer. The draw package could then include only the indices to the influencing lights. This would considerably reduce the needed amount of additional memory per draw package.

A different approach that has a constant memory requirement is the use of deferred shading as described in [14]. Deferred shading will generate a so called G-Buffer that stores all necessary information to shade a pixel in multiple screen buffers. This G-Buffer is then used to generate final pixel shading by invoking a pixel shader on each pixel a light touches. This is done for all lights in the scene to get the final image. This technique nicely decouples the geometric complexity of the scene from the light count but increases the memory usage considerably. Both the generation of the G-Buffer and the reading from the G-Buffer in the final shading step put pressure on the memory system.

A third approach is Forward+ which was implemented in *breach* and is described in the next Section.

9.1 Forward+ Light Culling

The first step in shading is to decide which lights affect a certain pixel on the screen. An algorithm very similar to [13] and [20] is used to solve this problem. All lights in the scene are added to a list in GPU memory just like the draw packages. A depth-only pre-pass is rendered to reduce the pixel shading cost of the final shading and to supply us with a depth buffer that is used in the next steps. After this the screen is split into tiles of 16×16 pixels and a compute shader is invoked for each of these tiles. Thes shader computes the minimum and maximum depth of a tile using the depth buffer and intersects all lights in the list with this bounding volume. The

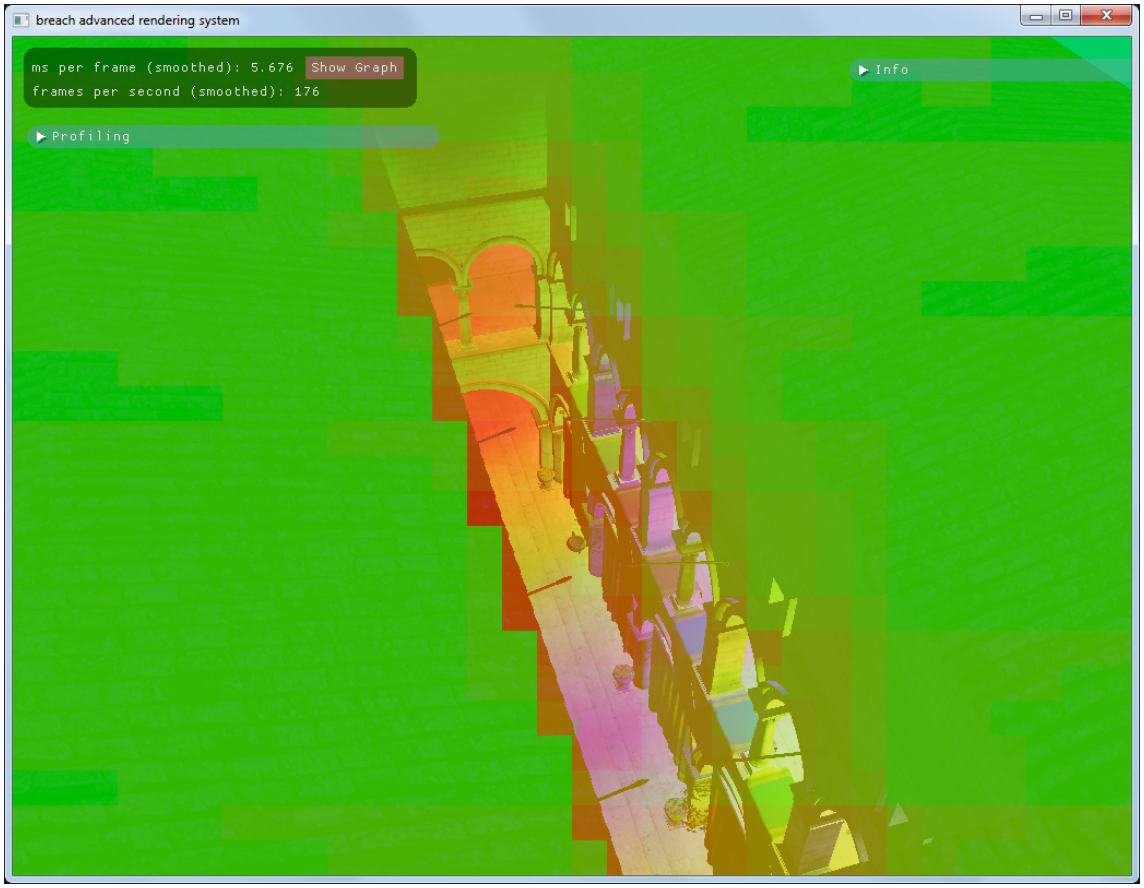

Figure 9: Light tiles visualized (green: no light, red: multiple lights affecting the tile)

index of each intersecting light is then stored in a linear buffer that is initialized with enough storage for 16 lights per tile.

The pixel shader can then easily access this light list for the corresponding tile when drawing the scene and use the stored index to load the corresponding light data. Figure 9 shows a debug visualization of the Sponza scene with 20 lights. Green tiles are not influenced by any light, the more redish the tiles become the more lights are affecting the tile.

9.2 Shadow Mapping

Shadow mapping, first described in [22], is a widespread technique for rendering shadows. It works by rendering the minimal depth (or distance) of the scene as seen from a virtual camera at a lights position. When drawing a pixel this distance is compared to the actual distance of the pixel from the light position. When the distance stored in the depth map is smaller than the distance computed for the

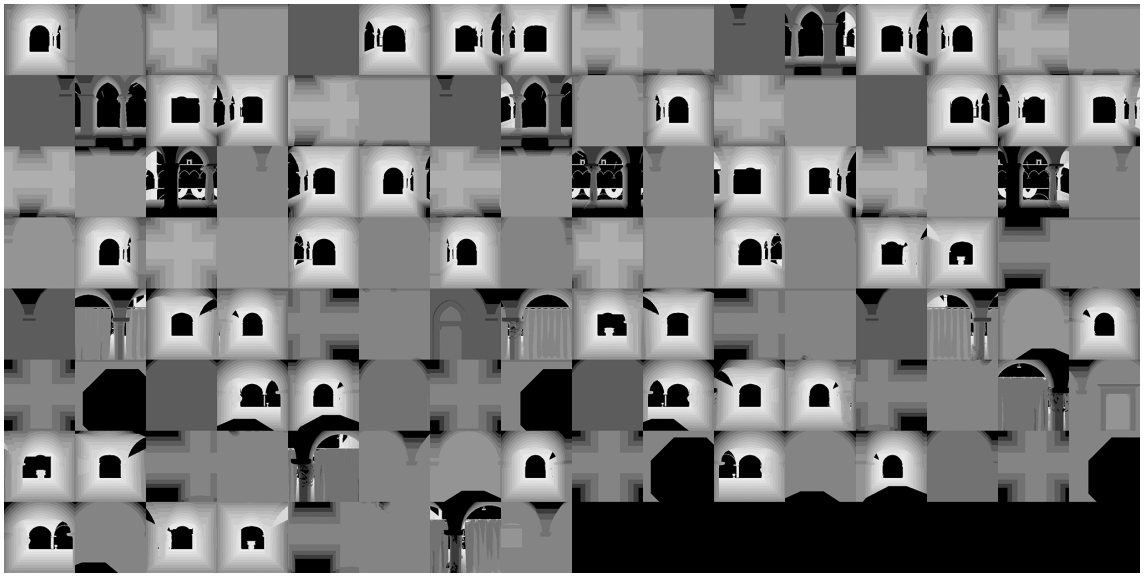

Figure 10: The atlas used to store the depth maps for shadow mapping.

pixel, some object obstructed the view from the light to the pixel. Therefore the pixel is not reached by the light and should lie in shadow.

Similar to the shading techniques described before some constraints have to be considered when using shadow mapping with the proposed rendering algorithm. As all shading is done in one draw call, all shadow maps for all visible lights have to be accessible to the pixel shader.

To be able to access all shadow maps from a shader without state changes a texture atlas is employed. Similar to the texture atlases described in Section 7 a single depth texture is filled with shadow maps from all lights. The part of the shadow map that is reserved for a specific light can either be assigned statically or dynamically. For simplicity it was decided to use a static assignment strategy in *breach*, where each light gets an index describing the position of the light in the shadow map. Only a fixed amount of shadow maps is updated every frame to spread the cost for generation among multiple frames.

Figure 10 shows an example of the texture atlas used to store all depth maps for shadow casting lights.

10 Results

The following section examines the performance characteristics of the proposed algorithm and compares it to a baseline renderer.

10.1 Default Renderer

To compare the results from the newly proposed rendering algorithm a second rendering path was implemented. This second *default* renderer uses a one draw call per object design for rendering. To ease development most of the resource management system is shared between both implementations. This includes the texture and material systems which allows the default renderer to use indexed materials as well. The core differences between the two rendering implementations lie in the draw submission stage. The draw submission for the *packaged* renderer is done using draw package queues, while the default renderer draws one object at a time. The packaged renderer performs view culling as part of the index expansion. To achieve the same rendering load submitted to the GPU a CPU implementation of the culling algorithm is used in the default renderer.

10.2 Testing Environment

10.2.1 Hardware

All scenes have been tested on five different hardware configurations. Two of them were laptop computers, the three others were desktop computers. Table 1 shows the hardware of all systems.

Name	**CPU**	**RAM**	**GPU**	**GPU RAM**	**OS**
Laptop 0	Intel Core i5-5200U	4012 MB	AMD Radeon R7 M260	2104 MB	Win 7 x64
Laptop 1	Intel Core i5-3317U	3981 M B	NVIDIA GeForce GT 620M	973 MB	Win 10 x64
Desktop 0	AMD Phenom 9650	3199 MB	AMD Radeon HD 5850	1015 MB	Win 7 x86
Desktop 1	AMD Phenom II X4 965	16382 MB	NVIDIA GeForce GTX 970	4008 MB	Win 10 x64
Desktop 2	Intel Core i5-3570K	16316 MB	AMD Radeon R9 290	4075 MB	Win 10 x64

Table 1: Hardware configuration used for testing.

10.2.2 Software

A special benchmark mode was added to the *breach* engine. This mode runs through a supplied list of scenes and measuring the rendering performance. The implemented shadow mapping in *breach* is quite performance heavy, therefore it was decided to disable it in order to allow low end hardware to participate in the testing. Every scene is drawn using the default renderer for five seconds, then the renderer is switched and drawing is resumed for another five seconds. The amount of frames rendered is recorded for both runs. From this number the average frame computation time for a scene is computed by dividing the number of frames by the time. The resulting frame time for each scene and renderer is saved in addition to measuring the average time of some CPU and GPU performance counters. These counters are implemented using the profiler discussed in Section 5. Table 2 shows the implemented counters and the part of code they measure. All performance information along with the system specifications is then stored in a JSON file for later review.

Name	**Type**	**Description**
World::Draw	CPU	CPU time needed to submit all objects to the renderer.
Renderer::Render Frame	CPU	CPU time used to submit all draw commands to the GPU.
Renderer::Render Prepass	GPU	GPU time used to draw the pre-pass.
Renderer::Clear Light Grid	GPU	GPU time needed to clear the light grid.
Renderer::Fill Light Grid	GPU	GPU time used to add all lights to the light grid.
Renderer::Forward Rendering	GPU	GPU time needed for the actual rendering and shading of all objects.
Renderer::Submit Indizes	GPU	GPU time used by the compute shader to expand all indices.

Table 2: Performance counters and the code they measure.

10.2.3 Scenes

All in all 22 scenes are tested in the benchmark.

Two of the scenes are variations of the widely used *Sponza* scene. The first is the Sponza version provided by Crytek [11] containing a single light shown in Figure 11. The second is the same scene with additional lights. The *Sponza Lights* scene contains 20 dynamic lights and is primarily used to test the light culling implementation. This scene can is shown in Figure 12

The remaining 20 scenes are synthetic test scenes generated using a tool developed as part of this thesis. The test data generation tool creates scenes containing a fixed amount of textures, materials, meshes and objects. It was used to create scenes containing 128 different materials (100 in the case of 100 objects per scene), with

[11] http://www.crytek.com/cryengine/cryengine3/downloads

Figure 11: The Crytek Sponza scene.

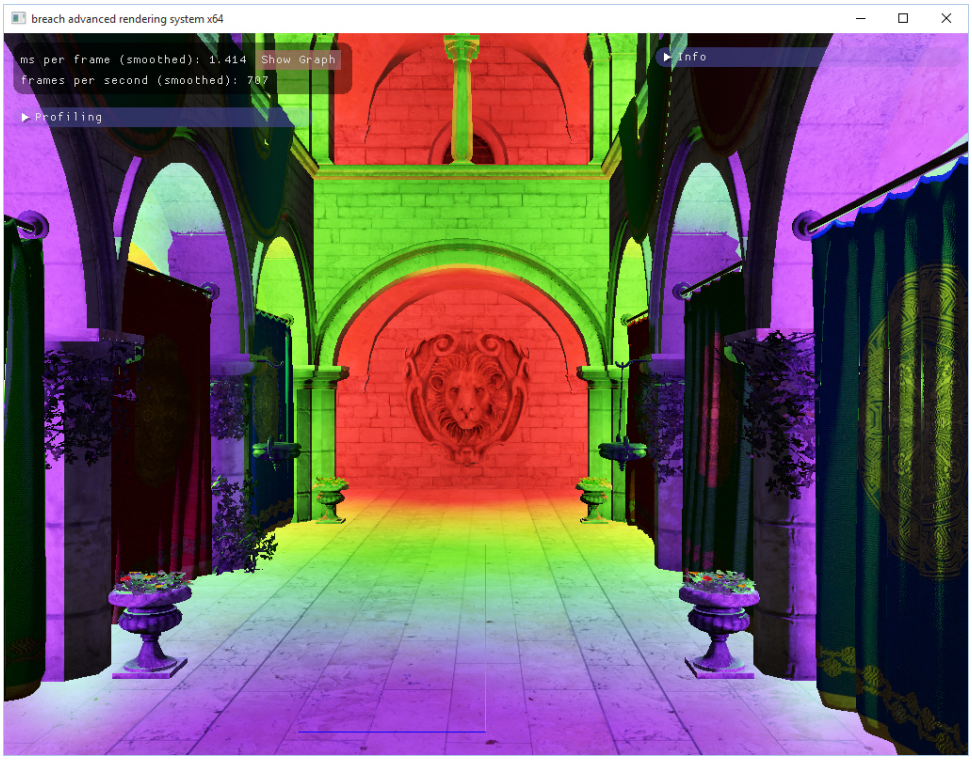

Figure 12: The Crytek Sponza scene with 20 lights.

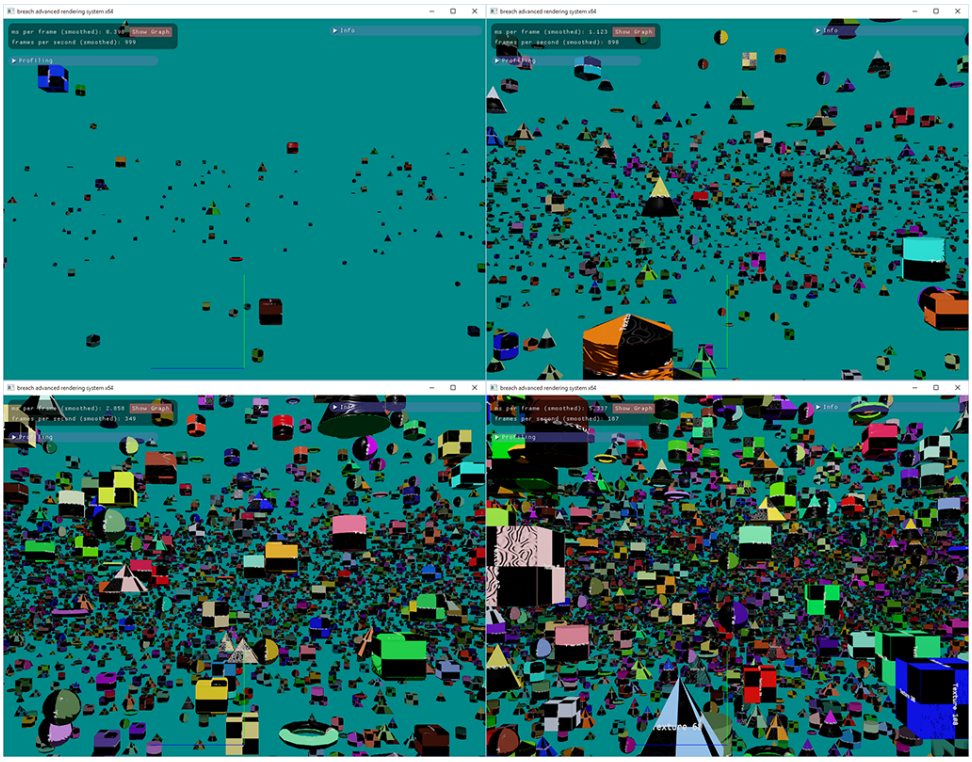

Figure 13: Four of the synthetic scenes.

unique textures and normal maps. The object count of the synthetic scenes rises in five levels (100, 1000, 10000, 32000, 64000). Object count and mesh count are decoupled. Meaning that for each different object count a version of the scene with 10, 100, 1000 and 10000 unique meshes is created resulting in a total of 20 different scenes with rising complexity. Table 13 shows a list of all scenes and their respective parameters. Note that scenes 1, 2 and 3 are equal as are scenes 6 and 7 because the object count in this scenes caps the mesh count.

10.3 Performance

10.3.1 Sponza

We will begin the performance discussion by analyzing the two Sponza scenes.

Figure 14 shows an overview of the average frame time per hardware configuration and an average over all measured hardware. The difference between both tested renderers is generally pretty small. Because the CPU workload for both the Sponza and the Sponza Lights scene is similar, the computation for the additional lights is very cheap CPU wise, the data indicates that both renderers are GPU limited in this

Name	Object Count	Mesh Count	Material Count	Triangle Count
test_0	100	10	10	9200
test_1		100	100	
test_2		100	100	
test_3		100	100	
test_4	1000	10	128	92000
test_5		100		
test_6		1000		
test_7		1000		
test_8	10000	10	128	920000
test_9		100		
test_10		1000		
test_11		10000		
test_12	32000	10	128	2944000
test_13		100		
test_14		1000		
test_15		10000		
test_16	64000	10	128	5888000
test_17		100		
test_18		1000		
test_19		10000		

Table 3: Synthetic scenes and their parameters.

case. Also evident from the data is the higher base cost of the packaged renderer. As many of the computations are done on the GPU the faster the GPU the lower the performance difference. In the case of Desktop 1, where a relatively old CPU is combined with a modern GPU, the packaged renderer is even faster by about 0.2 milliseconds in both scenes.

The average of all hardware configurations shows that as long as there is enough idle GPU time available, the packaged renderer is not incurring additional overhead. When the overall GPU demand is higher, as is the case in the Sponza Lights scene, the additional GPU overhead caused by expanding the vertices and manually loading them in the vertex shader will slow down the overall rendering.

10.3.2 Synthetic

The synthetic scenes have two main parameters: the object count and the mesh count. The mesh count describes how many different meshes are used in the scene, while the object count defines the amount of objects in the scene. A scene with an object count of 10000 and a mesh count of 1000 contains 10000 independently moving objects where every mesh is used 10 times in the scene.

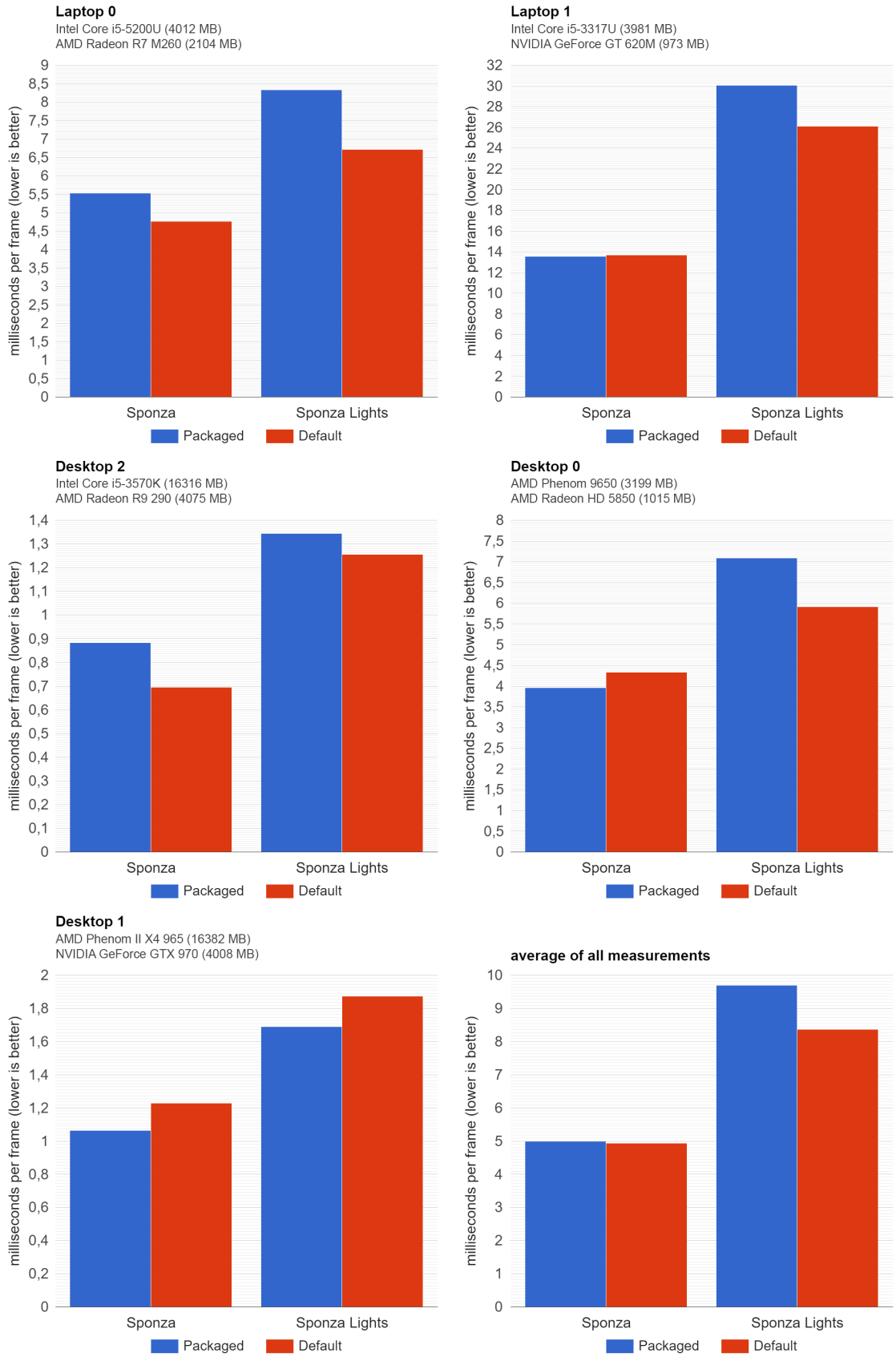

Figure 14: Results for the Sponza scenes.

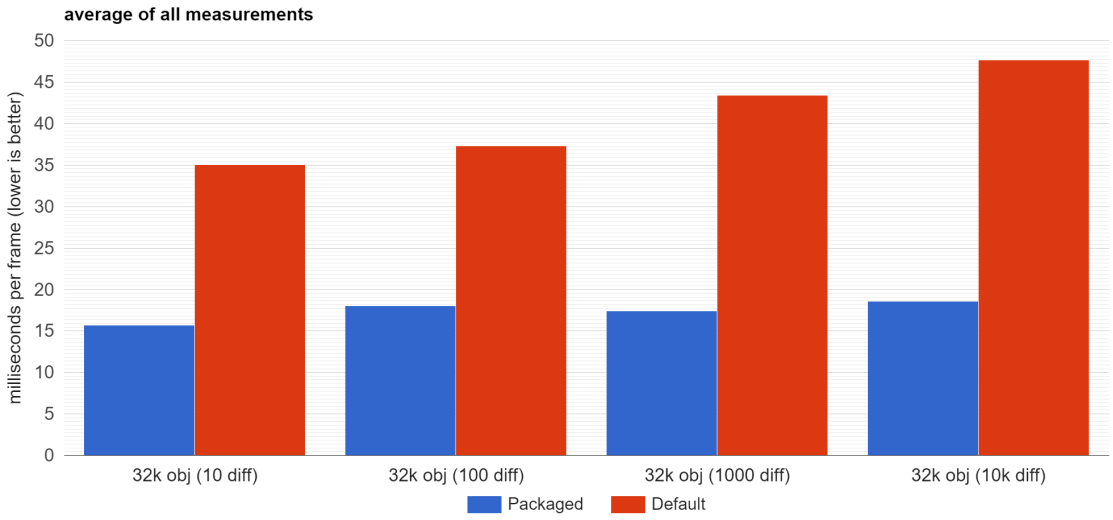

Figure 15: Synthetic scenes with 32000 objects.

Figure 15 shows the performance decrease when rendering 32000 objects with a variable amount of meshes. While the packaged renderer shows no significant change in render time when going from 10 to 10000 different meshes, the default renderer does. Here the renderer exhibits a negative effect on the performance when increasing the mesh count. It is likely that the storage of the individual meshes in separate vertex and index buffer is responsible for this behaviour. Unlike the packaged renderer, where all meshes are tightly packed into one area of GPU memory, the meshes in the default renderer can be scattered in the memory resulting in cache misses when many different meshes are used. The two renderers use separate ways to store the data because the vertex formats used to store the meshes for each renderer were different at the beginning of the development of *breach*. As Figure 16 shows the overall results indicates the same decrease in performance as Figure 15. To minimize the effect of this *historic* decision the remaining analysis here was done using only the results of using 10 different meshes.

The results shown in Figure 17 indicate that on average for object counts above 1000 objects the packaged renderer is an improvement over the default renderer. Up to that point the overhead of transferring the render queue and expanding indices is bigger than the gain from reduced CPU usage. Systems with a high GPU to CPU performance ratio like the desktop computers and Laptop 0 exhibit gains already with 1000 objects, while Laptop 1 is GPU limited up to 32000 objects. Above 1000 objects all systems benefit from packaged rendering. Improvements range from 166% on Laptop 1 to 528% of the default renderer frame throughput on Desktop 0. On

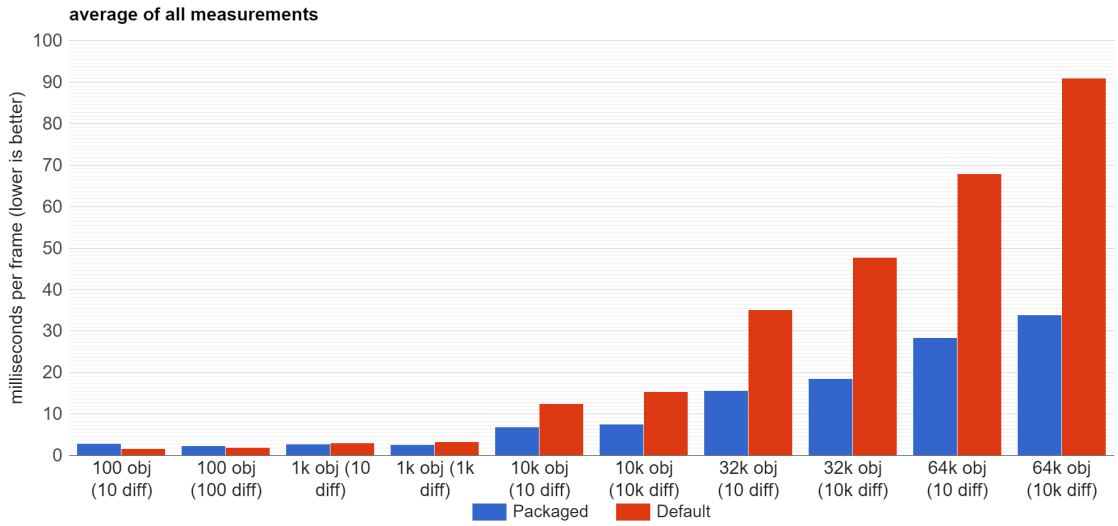

Figure 16: Comparison of results from using 10 meshes to 10000 meshes.

average rendering with the packaged rendering is more than twice as fast in the tested scenes.

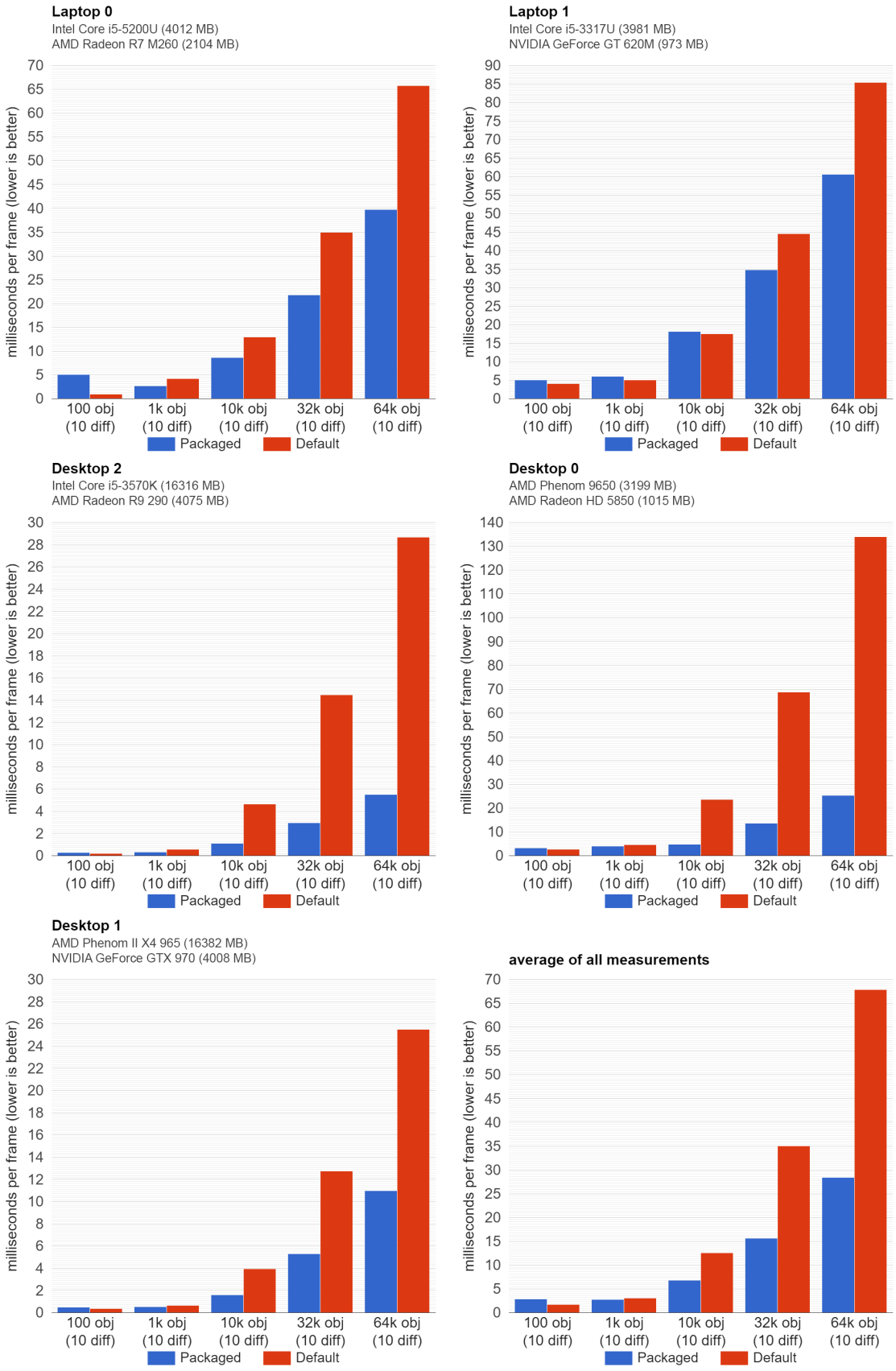

Figure 17: Results from drawing 100 to 64000 objects using 10 meshes.

11 Conclusion

As shown in the last section the improvements gained from the proposed algorithm are depending on both scene complexity and hardware. Simple scenes with a low object but high triangle count will not benefit from the reduced CPU usage as the main pressure is already on the GPU. The overhead of the additional GPU calculations for expanding indices and decoding vertices will therefore reduce the overall system performance in most cases.

In scenes with a high object count however the proposed solution can significantly improve performance as long as the application is not inherently GPU bound. This could happen either by slow graphics hardware or heavy calculations already done on the GPU.

A potential application of the algorithm is the use in editing tools. In most end-user applications static objects or objects sharing the same transformations will be baked together to reduce draw calls. In editor applications however individual objects have to stay separated to manipulate them. Therefore the proposed algorithm can be used to improve the overall performance of such applications.

Although new graphics APIs like Direct3D 12 and Vulkan are targeting the reduction of the overhead presented in this thesis, it is still valuable to know how to reduce the API overhead of legacy APIs. For once the new APIs will not be available on all devices supporting the old API and they will likely benefit as well from techniques similar to the presented.

The major constraint of this thesis was the use of the Direct3D 11 API. Future research in this direction should include new APIs and test if they can benefit from a similar technique, and how the proposed algorithm would be adapted to additional API features.

Additional Code Listings

Listing 11: Implementation of the geometry conditioner using the assimp library.

```
#include <assimp/Importer.hpp>     // C++ importer interface
#include <assimp/scene.h>       // Output data structure
#include <assimp/postprocess.h>   // Post processing flags

namespace breach {
  namespace resources {

    breach_external bool
    ConditionGeometry(const char* inputPath, SDL_RWops* output)
    {
      if (!inputPath || !output) {
        return false;
      }

      Assimp::Importer importer;
      const aiScene* scene = importer.ReadFile(inputPath, aiProcess_CalcTangentSpace |
            aiProcess_Triangulate | aiProcess_JoinIdenticalVertices | aiProcess_MakeLeftHanded ↩
                |
            aiProcess_PreTransformVertices | aiProcess_GenSmoothNormals | aiProcess_FlipUVs |
            aiProcess_SortByPType;

      // If the import failed, report it
      if( !scene) {
        SDL_Log("Error while initializing asset importer: %s", importer.GetErrorString());
        return false;
      }

      // Fill a single mesh from the whole scene.
      std::vector<uint32>* indizes = new std::vector<uint32>();
      std::vector<ConditionedGeometryDataVertex>* vertices =
      new std::vector<ConditionedGeometryDataVertex>();

      int meshCount = scene->mNumMeshes;
      for (int i = 0; i < meshCount; ++i) {
        aiMesh* mesh = scene->mMeshes[i];
        int currentVertexIndex = (int)vertices->size();
        uint vertexCount = mesh->mNumVertices;
        for (uint v = 0; v < vertexCount; ++v) {
          ConditionedGeometryDataVertex vertex;
          aiVector3D pos = mesh->mVertices[v];
          vertex.position = glm::vec3(pos.x, pos.y, pos.z);

          if(mesh->HasNormals()) {
            aiVector3D normal = mesh->mNormals[v];
            vertex.packedNormalXY = glm::packHalf2x16(glm::vec2(normal.x, normal.y));
            vertex.packedNormalZmaterialIndex= glm::packHalf2x16(glm::vec2(normal.z, 0));
          }
          if(mesh->HasTextureCoords(0)) {
            aiVector3D uv = mesh->mTextureCoords[0][v];
            vertex.packedTexCoord = glm::packHalf2x16(glm::vec2(uv.x, uv.y));
          }
          vertices->push_back(vertex);
        }

        uint indexCount = mesh->mNumFaces;
        for (uint f = 0; f < indexCount; ++f) {
          aiFace face = mesh->mFaces[f];
          for (int c = 0; c < 3; ++c) {
            // only supports triangles
            indizes->push_back(face.mIndices[c] + currentVertexIndex);
          }
        }

      }

      ConditionedGeometryDataHeader geoDataHeader;
      geoDataHeader.indexCount = (uint32)indizes->size();
      geoDataHeader.vertexCount = (uint32)vertices->size();
      geoDataHeader.materialCount = 1;

      SDL_RWwrite(output, &geoDataHeader, sizeof(geoDataHeader), 1);
      SDL_RWwrite(output, vertices->data(), sizeof(ConditionedGeometryDataVertex) *
                                            vertices->size(), 1);
      SDL_RWwrite(output, indizes->data(), sizeof(uint32) * indizes->size(), 1);

      delete indizes;
      delete vertices;

      return true;
    }
  }
}
```

Listing 12: HLSL code to expand the packet indices.

```
struct DrawPacket
{
  matrix worldTransform;
  uint startVertex;
  uint startIndex;
  uint indexCount;
  uint materialIndex;
  uint geometryIndex;
};

struct AABB
{
  float3 minVal;
  float3 maxVal;
};

struct AABBIndexed
{
  float3 center;
  float3 halfExtent;
};

cbuffer PerView : register(b0)
{
  matrix ViewProjection;
  matrix ViewProjectionInv;
  float4 ViewPlanes[6];
  float3 ViewPosition;
  float ViewRadius;
};

cbuffer PerRender : register(b1)
{
  int MaxDrawPacket;
  int MinDrawPacket;
  int MaxVertices;
};


StructuredBuffer<DrawPacket> drawPackets: register (t0);
StructuredBuffer<uint> indicesIn: register (t1);
StructuredBuffer<AABB> aabbIn: register (t2);

RWByteAddressBuffer indicesOut: register (u0);
RWByteAddressBuffer drawArgs: register (u1);

#ifndef NoAABB
RWStructuredBuffer<AABBIndexed> aabbOut: register (u2);
#endif

//~256 for desktop // ~32 for mobile
static const int TGSize = 256;

groupshared DrawPacket packet;
groupshared uint startInOutput;
groupshared bool isVisible;

bool IsVisible(AABBIndexed aabb)
{
  for (int i = 0; i < 6; i++) {
    float3 absPlane = abs(ViewPlanes[i].xyz);
    float d = dot(aabb.center, ViewPlanes[i].xyz);
    float r = dot(aabb.halfExtent, absPlane);

    if (d + r < -ViewPlanes[i].w) {
      return false;
    }
  }
  return true;
}

// Code to use if no frustum culling is active.
#ifdef NoFrustumCull
[numthreads(TGSize, 1, 1)]
void CS(uint3 Gid : SV_GroupID, uint3 GTid : SV_GroupThreadID)
{
  uint id = MinDrawPacket + Gid.x + Gid.y * 65535;
  uint threadIndex = GTid.x;

  // Load aabb and test if visible.
  if (threadIndex == 0) {
    packet = drawPackets[id];

    drawArgs.InterlockedAdd(0, packet.indexCount, startInOutput);
  }

  GroupMemoryBarrierWithGroupSync();

  // Copy indizes.
  for (uint i = threadIndex; i < packet.indexCount; i += TGSize) {
    uint inI = packet.startIndex + i;

    // * 4 because we use a byte indexed buffer
    uint outI = (startInOutput + i) * 4;
```

```
    uint index = indicesIn[inI];
    indicesOut.Store(outI, ((index << 16) & 0xFFFF0000) | (id & 0x0000FFFF));
  }
}

#else
// Code to use if frustum culling is active.
[numthreads(TGSize, 1, 1)]
void CS(uint3 Gid : SV_GroupID, uint3 GTid : SV_GroupThreadID)
{
  uint id = MinDrawPacket + Gid.x + Gid.y * 65535;
  uint threadIndex = GTid.x;

  // Load aabb and test if visible.
  if (threadIndex == 0) {
    packet = drawPackets[id];

    // Transform AABB vertices.
    AABB aabbLocal = aabbIn[packet.geometryIndex];
    float4 aabbVertices[8];
    aabbVertices[0] = float4(aabbLocal.minVal.x, aabbLocal.minVal.y, aabbLocal.minVal.z, 1);
    aabbVertices[1] = float4(aabbLocal.minVal.x, aabbLocal.minVal.y, aabbLocal.maxVal.z, 1);
    aabbVertices[2] = float4(aabbLocal.maxVal.x, aabbLocal.minVal.y, aabbLocal.minVal.z, 1);
    aabbVertices[3] = float4(aabbLocal.maxVal.x, aabbLocal.minVal.y, aabbLocal.maxVal.z, 1);
    aabbVertices[4] = float4(aabbLocal.minVal.x, aabbLocal.maxVal.y, aabbLocal.minVal.z, 1);
    aabbVertices[5] = float4(aabbLocal.minVal.x, aabbLocal.maxVal.y, aabbLocal.maxVal.z, 1);
    aabbVertices[6] = float4(aabbLocal.maxVal.x, aabbLocal.maxVal.y, aabbLocal.minVal.z, 1);
    aabbVertices[7] = float4(aabbLocal.maxVal.x, aabbLocal.maxVal.y, aabbLocal.maxVal.z, 1);

    // Compute new AABB
    float3 valMin = 10000000.0f, valMax = -10000000.0f;
    for (int i = 0; i < 8; ++i) {
        float3 tmpPos = mul(packet.worldTransform, aabbVertices[i]).xyz;
        valMin = min(valMin, tmpPos);
        valMax = max(valMax, tmpPos);
    }

    AABBIndexed aabb;
    aabb.center = (valMin + valMax) * 0.5f;
    aabb.halfExtent = (valMax - aabb.center);

#ifndef NoAABB
    // This is used to display the AABB in debug view.
    aabbOut[id] = aabb;
#endif

    // Check visibility.
    isVisible = IsVisible(aabb);
    if (isVisible) {
        drawArgs.InterlockedAdd(0, packet.indexCount, startInOutput);
    }
  }

  GroupMemoryBarrierWithGroupSync();

  // Copy indizes.
  if (isVisible) {
    for (uint i = threadIndex; i < packet.indexCount; i += TGSize) {
      uint inI = packet.startIndex + i;
       // * 4 because we use a byte indexed buffer
      uint outI = (startInOutput + i) * 4;
      uint index = indicesIn[inI];
      indicesOut.Store(outI, ((index << 16) & 0xFFFF0000) | (id & 0x0000FFFF));
    }
  }
}
#endif
```

Listing 13: HLSL code to generate the light lookup grid.

```
cbuffer PerView : register(b0)
{
  matrix ViewProjection;
  matrix ViewProjectionInv;
  float4 ViewPlanes[6];
  float3 ViewPosition;
  float ViewRadius;
};

struct LightTransform
{
  float3 position;
  float maxRadius;
};

cbuffer LightBuffer : register(b1)
{
  uint lightCount;
  uint unused;
  uint screenWidth;
  uint screenHeight;
};
Texture2D depthMap : register(t0);
StructuredBuffer<LightTransform> lights : register(t1);

static const int TGSizeX = 16;
static const int TGSizeY = 16;
static const int TGCount = TGSizeX * TGSizeY;

RWTexture2D<uint> lightGrid2D : register(u0);
RWStructuredBuffer<uint> lightIndizes : register(u1);

groupshared int2 minTile;
groupshared int maxDepthTileInt;
groupshared float maxDepthTileFloat;
groupshared uint sharedFlatIndex;
groupshared uint sharedOutLightCount;

groupshared float4 tilePlane[5];

float4 PlaneFromPoints(float3 p0, float3 p1, float3 p2)
{
  float3 v = p1 - p0;
  float3 u = p2 - p0;
  float3 n = normalize(cross(v, u));
  return float4(n, -dot(n, p0));
}

float DistanceFromPlane(float4 plane, float3 p)
{
  return dot(plane.xyz, p) + plane.w;
}

[numthreads(TGSizeX, TGSizeY, 1)]
void CS(uint2 Gid : SV_GroupID,
        uint2 threadIndex : SV_GroupThreadID)
{
  if(threadIndex.x + threadIndex.y == 0) {
    minTile.x = Gid.x * TileSizeX;
    minTile.y = Gid.y * TileSizeY;
    maxDepthTileInt = 0x80000000;
  }

  GroupMemoryBarrierWithGroupSync();

  for (uint x = threadIndex.x; x < TileSizeX; x += TGSizeX) {
    for (uint y = threadIndex.y; y < TileSizeY; y += TGSizeY) {
      int3 loadCoordDepth = int3(minTile.x + x,
                                 minTile.y + y,
                                 0);
      int depthInt = asint(depthMap.Load(loadCoordDepth).r);
      InterlockedMax(maxDepthTileInt, depthInt);
    }
  }

  GroupMemoryBarrierWithGroupSync();

  if(threadIndex.x + threadIndex.y == 0) {
    sharedOutLightCount = 0;
    maxDepthTileFloat = asfloat(maxDepthTileInt);

    float tileCountX = ceil((float)screenWidth / TileSizeX);
    float tileCountY = ceil((float)screenHeight / TileSizeY);
    sharedFlatIndex = (Gid.y * tileCountX + Gid.x) * MaxLightsPerTile;

    float tileSizeClipX = 2.0f / tileCountX;
    float tileSizeClipY = 2.0f / tileCountY;
    float4 bl = float4(((float)Gid.x / tileCountX) * 2.0f - 1.0f,
                       1.0f - ((float)Gid.y / tileCountY) * 2.0f,
                       maxDepthTileFloat,
                       1.0f);

    float4 tl = float4(bl.x, bl.y - tileSizeClipY,
                       maxDepthTileFloat, 1.0f);
```

```
    float4 tr = float4(tl.x + tileSizeClipX, tl.y,
                       maxDepthTileFloat, 1.0f);
    float4 br = float4(tr.x, bl.y, maxDepthTileFloat, 1.0f);

    tl = mul(tl, ViewProjectionInv); tl.xyz /= tl.w;
    tr = mul(tr, ViewProjectionInv); tr.xyz /= tr.w;
    bl = mul(bl, ViewProjectionInv); bl.xyz /= bl.w;
    br = mul(br, ViewProjectionInv); br.xyz /= br.w;

    tilePlane[0] = PlaneFromPoints(tl.xyz, tr.xyz, ViewPosition); // Top
    tilePlane[1] = PlaneFromPoints(tr.xyz, br.xyz, ViewPosition); // Right
    tilePlane[2] = PlaneFromPoints(br.xyz, bl.xyz, ViewPosition); // Bottom
    tilePlane[3] = PlaneFromPoints(bl.xyz, tl.xyz, ViewPosition); // Left
    tilePlane[4] = PlaneFromPoints(tl.xyz, bl.xyz, tr.xyz); // Far
  }

  GroupMemoryBarrierWithGroupSync();

  uint flatThreadIndex = threadIndex.y * TGSizeX + threadIndex.x;

  for (uint i = flatThreadIndex; i < lightCount; i += TGCount) {
    LightTransform tLight = lights[i];
    bool visible = true;
    for (uint f = 0; f < 5; f++) {
      float distance = DistanceFromPlane(tilePlane[f], tLight.position);
      if (distance < -tLight.maxRadius) {
        visible = false;
        break;
      }
    }
    if (visible) {
      if(sharedOutLightCount >= MaxLightsPerTile)
        break;

      uint elementIndex = 0;
      InterlockedAdd(sharedOutLightCount, 1, elementIndex);

      lightIndizes[sharedFlatIndex + elementIndex] = i;
    }
  }

  GroupMemoryBarrierWithGroupSync();
  if(threadIndex.x + threadIndex.y == 0) {
    lightGrid2D[Gid] = sharedOutLightCount;
  }

}
```

Bibliography

[1] AMD. Mantle White Paper, 2013. URL http://www.amd.com/Documents/Mantle_White_Paper.pdf.

[2] Dan Baker. Firaxis LORE And other uses of D3D11, 2011. URL http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Firaxis LORE.pps.

[3] Sean T. Barrett. stb libs, 2014. URL https://github.com/nothings/stb.

[4] Jeff Bolz. OpenGL bindless extensions, 2009.

[5] Jeff Bolz, Pat Brown, Chris Dodd, Mark Kilgard, and Eric Werness. Gl_nv_shader_buffer_load, 2013. URL https://www.opengl.org/registry/specs/NV/shader_buffer_load.txt.

[6] Jeff Bolz, Pat Brown, and Daniel Koch. Gl_nv_bindless_texture, 2013. URL https://www.opengl.org/registry/specs/NV/bindless_texture.txt.

[7] Jeff Bolz, Pat Brown, Graham Sellers, Pierre Boudier, and Daniel Koch. Gl_arb_bindless_texture, 2013. URL https://www.opengl.org/registry/specs/ARB/bindless_texture.txt.

[8] Christophe Riccio and Sean Lilley. Introducing the programmable vertex pulling rendering pipeline. In Wolfgang Engel, editor, *GPU Pro 4*, pages 21–37. CRC Press, 2013.

[9] Omar Cornut. ImGui, 2014. URL https://github.com/ocornut/imgui.

[10] Cass Everitt and John McDonald. Beyond porting, 2014. URL http://media.steampowered.com/apps/steamdevdays/slides/beyondporting.pdf.

[11] G-Truc Creation. Opengl hardware matrix, 2015. URL http://www.g-truc.net/doc/OpenGL 4 Hardware Matrix.pdf.

[12] Alexander Gessler, Thomas Schulze, Kim Kulling, and David Nadlinger. Open asset import library, 2015. URL http://assimp.sourceforge.net.

[13] Takahiro Harada, Jay McKee, and Jason C. Yang. Forward+: Bringing Deferred Lighting to the Next Level. In Carlos Andujar and Enrico Puppo, editors, *Eurographics 2012 - Short Papers*. The Eurographics Association, 2012. doi: 10.2312/conf/EG2012/short/005-008.

[14] Shawn Hargreaves. Deferred shading: 6800 Leagues under the sea, 2004.

[15] Sebastien Hillaire. Improving Performance by Reducing Calls to the Driver. In Cozzi, Patrick and Riccio, Christophe, editors, *OpenGL Insights*, pages 353–364. A K Peters/CRC Press, 2012. doi: 10.1201/b12288-30.

[16] KHRONOS Group. Vulkan Overview, 2015. URL `https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf`.

[17] Sam Lantinga. SDL - Simple DirectMedia Layer. URL `https://www.libsdl.org`.

[18] Microsoft. DirectXTex texture processing library, 2014. URL `https://directxtex.codeplex.com`.

[19] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476628. URL `http://doi.acm.org/10.1145/1476589.1476628`.

[20] Ola Olsson and Ulf Assarsson. Tiled Shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251, 2011. doi: 10.1080/2151237X.2011.621761.

[21] Christophe Riccio. OpenGL Mathematics, 2014. URL `http://glm.g-truc.net/0.9.6/index.html`.

[22] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, August 1978. ISSN 0097-8930. doi: 10.1145/965139.807402. URL `http://doi.acm.org/10.1145/965139.807402`.

[23] Milo Yip. RapidJSON, 2014. URL `https://github.com/miloyip/rapidjson`.

Abstract

This master thesis concerns itself with the optimization of real-time computer graphics rendering performance. Wide spread graphics APIs like Direct3D 11 or OpenGL reduce the potential performance of graphics processing units by background validation and error checking for important API calls. To improve the utilization of current generation graphics hardware the overhead imposed by the application programming interface has to be reduced.

The thesis describes a specific way to process and layout data to reduce graphics API interaction to a minimum while still retaining the ability to draw fully dynamic scenes. All geometry, material and texture data is stored as array resources in GPU memory. Although this potentially increases the amount of memory needed, it enables the identification of a single GPU resource by the array index. At render time a queue of data packages containing such indices is generated. Each package is filled with all information necessary to render a specific object. To compact and cull this data a compute shader is invoked to process the input queue and output a single index buffer for rendering. This index buffer in combination with the resource arrays is then used to draw the whole scene by using a single draw call.

In addition to the overall architecture, the thesis also describes the implementation of this algorithm on modern hardware. The documentation of the rendering engine provided in this thesis details all necessary steps to implement the algorithm using the widespread Direct3D 11 graphics API.

Finally a second rendering architecture was implemented using multiple draw calls for rendering. The last part of the thesis compares the draw call efficiency of the two methods for different scenes showing the pros and cons of the proposed algorithm.

Zusammenfassung

Diese Masterarbeit beschäftigt sich mit der Optimierung von Echtzeit-Computergrafik. Viele weit verbreitete Grafikschnittstellen wie Direct3D 11 oder OpenGL reduzieren die erreichbare Leistung der Grafikhardware durch im Hintergrund durchgeführte Validierungsoperationen oder Ressourcen Management. Um die Auslastung des Grafikprozessors zu verbessern müssen die Leistungsverluste durch die Schnittstelle verringert werden.

Die Arbeit beschreibt einen spezielle Anordnung der Daten im Speicher, die es erlaubt die notwendige Interaktion zwischen der Applikation un der Grafikschnittstelle zu reduzieren. Alle Modelle, Materialien und Texturen werden als Array im Grafikspeicher abgelegt. Dadurch steigt zwar die erforderliche Menge an Grafikspeicher, ermöglicht jedoch die Identifikation einzelner Ressourcen über den Array-Index. Um die Szene zu rendern, wird eine Queue angelegt, deren Einträge alle Informationen enthalten, um ein bestimmtes Objekt darzustellen. Im nächsten Schritt wird ein Compute Shader verwendet, um alle Objekte aus der Queue zu entfernen, die nicht gesehen werden. Das Resultat dieses Compute Shaders ist ein Array mit Geometrie-Indizes. Diese Indizes können in Verbindung mit den Ressourcen-Arrays verwendet werden, um die gesamte Szenengeometrie in einem Schritt zu zeichnen.

Zusätzlich zum allgemeinen Algorithmus beschreibt diese Masterarbeit auch eine spezifische Implementation auf aktueller Hardware unter Zuhilfenahme der Direct3D 11 Grafikschnittstelle.

Weiters wurde ein zweiter Algorithmus implementiert der jedes Objekt einzeln zeichnet. Die Leistung beider Algorithmen wurde verglichen indem verschiedene Szenen mit beiden Implementationen auf verschiedener Hardware gezeichnet wurden.

I dedicate this thesis to my parents, who supported me on every step I took and made it possible for me to study computer science in the first place. I would also like to thank my girlfriend Marie for putting up with my terrible English when reading this work and cheering me up when I needed it the most. Special thanks go to Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs for patiently supervising this thesis and for letting me be part of his wonderful team at the Entertainment Computing Group. Finally I want to thank all friends and colleagues who supported me in the course of my studies and beyond.

Curriculum Vitae

Allgemein

Name	**Daniel Martinek**
Geburtsdatum	13.12.1987
Geburtsort	Schwaz, Austria
Kontakt	d.martinek@23volts.com

Bildung

2012-heute	**Universität Wien** Masterstudium Medieninformatik
2008-2012	**Universität Wien** Bachelorstudium Medieninformatik Abschluss: 22.10.2012
2002-2007	**HTL Jenbach** Wirtschaftsingenieurwesen Abschluss: 19.06.2007

Arbeitserfahrung

Okt. 2013-heute	**Universität Wien** Projektmitarbeiter "INTERACCT"
Nov. 2012-Okt. 2013	**Universität Wien** Projektmitarbeiter "Play the Net"

Publikationen

siehe nächste Seite

Publikationen

2015

K. Peters, F. Kayali, A. Lawitschka, M. Silbernagl, R. Matheus-Berr, D. Martinek, H. Hlavacs, INTERACCT: Remote Data Entry System with Game-Elements for young Leukaemia Patients, IEEE Healthcom 2015, Boston, USA, Oct. 14-17 2015.

R. Mateus-Berr, B. Brunmair, H. Hlavacs, F. Kayali, J. Kuczwara, S. Lehner, D. Martinek, M. Nebel, K. Peters, A. Reithofer, R. Woelfle, M. Silbernagl, M. Sprung, Co-Designing Avatars for Children with Cancer, LearnxDesign 2015: The 3rd International Conference for Design Education Researchers and PreK-16 Design Educators, Chicago, June 28-30 2015.

F. Kayali, K. Peters, J. Kuczwara, A. Reithofer, D. Martinek, R. Wölfle, R. Mateus-Berr, Z. Lehner, M. Silbernagl, M. Sprung, A. Lawitschka, H. Hlavacs, Participatory Game Design for the INTERACCT Serious Game for Health, Joint Conference on Serious Games (JCSG) 2015, Huddersfield, UK, 3-4 June 2015.

K. Peters, F. Kayali, A. Reithofer, R. Wölfle, R. Mateus-Berr, J. Kuczwara, Z. Lehner, A. Lawitschka, B. Brunmaier, D. Martinek, M. Silbernagl, H. Hlavacs, Serious Game Scores as Health Condition Indicator for Cancer Patients, Medical Informatics Europe (MIE) 2015, Madrid, Spain, May 27.-29 2015.

K. Peters, F. Kayali, A. Reithofer, R. Mateus-Berr, J. Kuczwara, A. Lawitschka, D. Martinek, M. Silbernagl, H. Hlavacs, Serious Game Scores as Health Condition Indicator for Cancer Patients, Medical Informatics Europe (MIE) 2015, Madrid, Spain, 27.-29.5.2015.

2014

F. Kayali, K. Peters, A. Reithofer, R. Mateus-Berr, Z. Lehner, D. Martinek, M. Sprung, M. Silbernagl, A. Lawitschka, H. Hlavacs, A Participatory Game Design Approach for Children After Cancer Treatment, ACE 2014 workshop Designing Systems for Health and Entertainment: what are we missing?, Funchal, Madeira, 11. Nov. 2014.

F. Kayali, G. Wallner, S. Kriglstein, G. Bauer, D. Martinek, H. Hlavacs, P. Purgathofer and R. Woelfle, A Case Study of a Learning Game about the Internet, GameDays 2014, Darmstadt, Germany, April 1-4 2014.

2013

Bauer, G., Martinek, D., Kriglstein, S., Wallner, G. & Wölfle, R., Digital game-based learning with "Internet Hero": A game about the internet for children aged 9–12 years, Context Matters!: Exploring and Reframing Games and Play in Context. Migutsch, K. (Hrsg.). Wien: New Academic Press, S. 148-161 14 S. 12, 2013.