Utrecht University

Department of Information and Computing Sciences

Master's Thesis

Optimizing Draw Call Batching Using Transient Data-Guided Texture Atlases

Jordy van Dortmont

ICA-3894479

Supervisors

B. Zalmstra BSc	dr. S.W.B. Prasetya	dr. ing. J. Bikker
Abbey Games	Utrecht University	Utrecht University

November 27, 2017

Abstract

Rendering a large number of 2D textures in real-time requires reducing the overhead of a large number of draw calls on the CPU caused by binding different textures when drawing. Texture atlases are used to avoid switching textures by packing textures into one larger texture before rendering. Graphics hardware APIs limit the size of a texture, so textures need to be partitioned into multiple atlases. Unfortunately, composing textures into atlases is performed manually by developers or artists with an educated guess and requires manually checking which texture switch breaks a draw call batch to improve batching. Manual composition of texture atlases is cumbersome, time-consuming and not optimal for large-scale and unpredictable use of textures. We automated the composition of atlases based on previously gathered texture rendering data to remove manual intervention and created transient texture atlases at run-time based on usage to optimize draw call batching. We applied our approach to four simulations and measured the number of draw calls, CPU frame time and GPU frame time. The number of draw calls is similar or less for data-guided texture atlases than for manually composed texture atlases. Transient data-guided texture atlases reduce the number of draw calls significantly for unpredictable use of textures, which leads to performance gains on the CPU.

Acknowledgements

First I want to thank Bas Zalmstra and Abbey Games for supporting me, allowing me to develop my Master's thesis project in their game engine and working together with me. I also want to thank my supervisor Wishnu Prasetya for his research guidance, support and evaluation and my second supervisor Jacco Bikker for reviewing and evaluating my Master's thesis. I want to express my gratitude to my parents for their unconditional support and encouragement not only during my thesis but also throughout my whole educational career. Finally, I would like to thank Laura Boer for her love and support.

Contents

Chapter 1

Introduction

Texture mapping is a common practice in 3D games that involves the wrapping and mapping of pixels onto a surface of a 3D model. Usually these pixels originate from a 2D image, a texture, that adds detail to a 3D surface. For example, a floor can be textured with an image of the top of a wooden plank. In 2D games however 3D models are replaced by 2D sprites. A sprite is a 2D entity that can be manipulated in the game, by rotation or translation for example. To visualize a sprite, a texture is used as a resource that can be shared between multiple sprites.

The geometry of a 3D model or a 2D sprite consists of primitives, such as triangles. When rendering a frame, the geometry is passed through the rendering pipeline, which is a sequence of steps to draw the geometry. This is done by submitting draw calls to the GPU. A draw call is a driver API function call for drawing one or more primitives. Not all primitives of the geometry have the same texture or shader, which is a program that runs for each unit (e.g. a pixel or a vertex). Each time primitives have a different texture than previously, the render pipeline state changes, because a different texture has to be bound. When the render pipeline state changes, a new draw call needs to be done for the primitives with the different texture. Typically a game has many different entities with different textures that need to be drawn in a frame, so many draw calls are issued when rendering a frame, which can severely impact CPU performance and make the frame time CPU-bound[6][25]. To improve performance primitives with the same render pipeline state are batched together. So drawing a frame with optimal performance consists of minimizing draw calls by batching primitives with the same render pipeline state. If a group of primitives does not have the previous render pipeline state the current batch is broken up into two batches, thus breaking the batch. Render pipeline state changes cannot be solely attributed to texture bindings, but also to binding a pixel shader, different texture formats or primitive indices for example. However, texture binding is one of the most common batch-breakers[26].

Texture atlases are used to reduce the number of times a different texture is bound[6][15][26]. A texture atlas contains multiple textures that can be drawn together in one draw call, because a texture atlas is stored in memory as one larger texture, the texture does not need to change between draw calls. An example of a texture atlas can be seen in Figure 1.1. When rendering a texture that was separate before, we look it up in the texture atlas. Optimally all textures that are drawn together in a frame reside in one single atlas. However, texture atlases also need to be loaded into GPU memory and the maximum texture dimensions are limited. In addition, loading a single atlas of which only a small number of textures is used, uses a lot of GPU memory, which leads to poor performance. If a texture atlas exceeds the maximum texture dimensions, a single atlas cannot be created, the textures need to be divided over multiple atlases. Multiple atlases do not have to be loaded into memory at the same time and can be unloaded anytime, when the containing textures are not in use anymore. Textures are usually packed into atlases by manual selection for atlases when building the assets. Unfortunately it can be hard to achieve optimal atlases, because the manual process of composing the texture atlases is cumbersome for large numbers of textures, the usage of textures might not be apparent and building texture atlases before running the game is not optimal due to the possibly varying and dynamic usage of textures that cannot be predetermined. For example when a player assembles a team of characters with different sets of textures, a different combination of those sets of textures can be used each time the player assembles a new team. Although it would reduce draw calls if the textures of all characters were in one atlas, they might not all fit and even when they would fit, GPU memory is wasted by loading textures of characters that are not going to be used. The assembly of the team by the player cannot be predetermined, so the characters may all need to be put into separate atlases to ensure consistent performance.

We aim to eliminate or at least decrease the need for manual composition of texture atlases while also reducing the number of draw calls and to provide a solution to improve draw call batching for varying, dynamic and emergent texture usage while running a game. Automating the composition of texture atlases is also necessary to solve the problem of improving draw call batching while running a game. By gathering data when rendering a game in previous runs, we can guide an automatic process of composing and packing textures into texture atlases before run-time and we can also guide batching improvements while running a game the next time. We aim to outperform the overhead that is caused by applying this approach through reducing the number of draw calls per frame significantly.

In the rest of this chapter we elaborate on how textures are stored and used to lay the groundwork for the approach in Chapter 2. Thereafter we dig deeper into our implementation in Chapter 3. To put our approach to the test we performed experiments of which the design is detailed in Chapter 4 and the results are shown in Chapter 5. We discuss the outcomes in Chapter 6, position and compare our approach in Chapter 7, and conclude this thesis in Chapter 8.

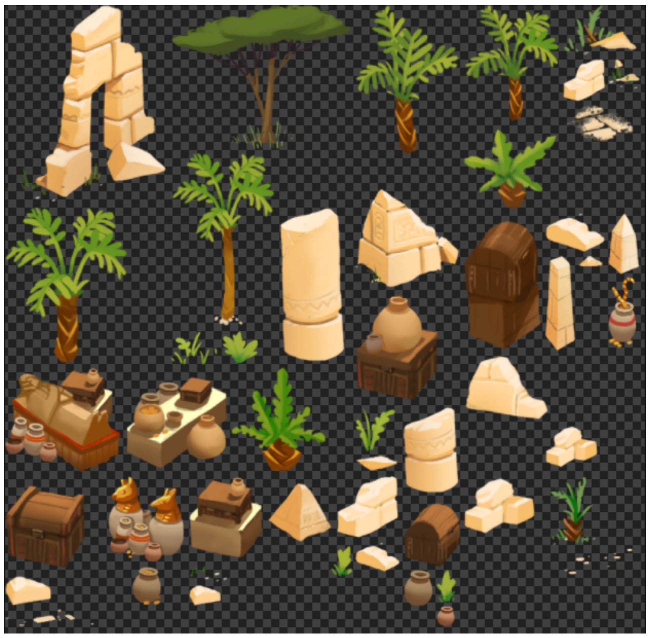


Figure 1.1: A texture atlas of Egyptian props used in Renowned Explorers: International Society[17].

1.1 Mipmapping

When an object is viewed from a distance, it is usually sufficient to render the texture of the object with less detail to avoid aliasing and unnecessarily detailed rendering[23]. Williams presents mipmaps as a parameterized pyramidal data structure[24] to optimize rendering a texture at a lower level of detail. A mipmap is parametrized by texture coordinates and by a mip level. Texture coordinates are on the two axes of a 2D texture, denoted by U and V and are used to look up a texel (texture pixel) in a texture. Each subsequent mip level changes the resolution of these axes. The top level is usually the most detailed and the bottom level the least detailed. The resolution of a texture is divided by half for each level in the pyramid, which diminishes the level of detail with each additional level in a mipmap. When rendering a texture at a small level of detail, for example 2 by 2 texels, all the texels of a texture of 512 by 512 texels have to be filtered and averaged into those 2 by 2 texels, this can become very costly at run-time. To prevent these unnecessary computations at run-time a mipmap is precomputed and contains the filtered and averaged resolutions on multiple levels in its pyramidal data structure. When the width and height of a texture are the same power of two, the texture can be stored very efficiently in memory and can easily be looked up at run-time with the texture coordinates and by computing a mip level based on the level of detail necessary.

All the mipmaps of a textures are placed into the same texture atlas to also reduce draw calls when a lower level of detail is sufficient. Unfortunately, this has some drawbacks for putting textures with a different number of mipmaps into the same atlas. If we were to put two textures with a different amount of mipmap levels into one texture atlas, the range of mipmap levels to use or the bias towards certain mipmap levels should be adjusted between rendering a texture with a different mipmap level than the previous texture, which would break the batch even though that was exactly what we were trying to avoid by putting the two textures in the same atlas. So the two options left are equalizing the number of mipmap levels of the textures or just putting them into a different texture atlas. For the sake of brevity and clarity a texture is used as a reference to all the mipmaps that a texture has instead of stating the mipmaps of that texture individually.

1.2 Texture compression

For rendering high-resolution textures block compression is used to reduce memory requirements. Texture block compression operates on blocks of 4 by 4 texels and compresses those blocks into a given compression format. Some compression formats include only RGB bits for minimum storage requirements, but others can also contain alpha bits. The dimensions of a texture are required to be multiples of 4 for block compression. The mipmaps of a texture are also stored with the texture and the dimensions of the mipmaps also have to be multiples of 4 if a texture needs to be compressed. This means that the texture dimensions need to be aligned based on the number of mipmap levels and the block size for compression. Decompression is implemented in graphics hardware and has no performance penalty. Block compression not only increases the number of high-resolution textures we can store in GPU memory, but also decreases bandwidth usage, which improves real-time rendering performance[22][23].

Texture atlases are also textures, so if we want to apply compression, their dimensions should be aligned. Only textures with the same compression format can be put into the same texture atlas, because an atlas can only have one compression format. For example when we have one texture with an alpha channel and one without, we cannot put the texture with the alpha channel into an atlas with a compression format without an alpha channel. However, we can put the texture without an alpha channel into a compression format with an alpha channel, but it increases the memory requirements for the texture originally without an alpha channel.

1.3 Texture packing

Texture packing is used to create texture atlases. Texture packing takes a list of textures as the input, an atlas composition, and arranges the textures in a texture with larger dimensions. Usually this is done as compactly as possible to reduce memory requirements and to fit as many textures as possible into the larger texture. Packing textures into atlases is a 2D bin packing optimization problem, which is NP-hard, so different approximation algorithms and heuristics can make a significant difference. A rectangle bin packing algorithm and heuristic can be used to pack textures as rectangles[14], although more advanced and memory sparing texture packing algorithms for arbitrary shapes also exist[15]. In a rectangle bin packing algorithm the textures are arranged by using the dimensions a texture is stored in, because a 2D texture is stored as a rectangle. Rectangle bin packing algorithms and heuristics come in a variety of space and time complexities, picking the right algorithm and heuristic for the job is a space-time trade-off[14].

The whole texture including the mipmap levels is packed into a texture atlas and even though the texture atlas should be packed as tightly as possible, some padding is necessary to avoid texture bleeding. Texture bleeding occurs when applying a texture filtering technique, which takes into account neighboring texels of a looked up texel during rendering, to reduce visual artifacts when a texel does not directly correspond to a pixel. This means that one texture can *bleed* into another neighboring texture. Each mipmap level of a texture atlas should also account for texture bleeding with padding, even at the lowest level, because each subsequent level decreases the padding by half. When using block compression the placement of the texture in the atlas should be aligned based on the number of mipmap levels and the block size to keep the textures from bleeding on lower mipmap levels and to keep texture coordinates consistent on smaller mipmap levels.

Chapter 2

Approach

In this chapter we explain the approach for gathering texture rendering data, automatically generating texture atlas compositions and guiding the creation of new texture atlases while running a game. The implementation of our approach provides more depth and insight into the specifics of our approach in Chapter 3.

2.1 Telemetry

When composing texture atlases out of textures, we want to know which texture to put into which atlas. We can make an informed guess by manually checking which textures break the batch frequently and which textures can be packed into the same texture atlas, but it would be much better if we automate this process. To automate the traditionally manual collection of this knowledge, data is gathered from rendering a game. We call this *atlas telemetry*. Atlas telemetry registers the beginning and the end of a frame and the batch that is sent to the GPU when issuing a draw call. When a draw call is issued, the bound textures are registered, because those textures are used to draw the batch. We also keep track of the render pipeline state and whether it has changed or not between batches, regardless of whether one or more different textures have been bound between draw calls or not. A render pipeline state change is the reason a batch needs to be broken down into multiple batches and multiple draw calls need to be issued. Setting a texture also changes the render pipeline state, to avoid this we use texture atlases. When the render pipeline state only changes because of the texture, we might be able to merge this batch with the previous batch if the textures bound for the previous batch are packed together into an atlas with the textures bound for the current batch. If this is true we make pairs of these textures and count the frequency at which this occurs. Unfortunately, we should not make pairs of all textures that satisfy this condition, because textures might have different storage formats or different mipmap levels, so we cannot put them in the same atlas. We make pairs of textures and only count occurrences, because the amount of data is unreasonably large when logging all frames of many runs of a game or keeping track of all paths, instead of just pairs, of sequenced textures over multiple draw call batches[1]. By storing these frequencies we know how many times a batch is broken and which textures break the batch.

2.2 Telemetry data processing

We can extract compositions of the atlases from the data gathered by the atlas telemetry. We process the pairs of the textures and whenever a pair exists we put both textures into the same atlas composition. However, a texture atlas composition might not be enough to create texture atlases optimized for draw call batching if the textures in the composition do not fit into one texture atlas, so we also need a way to partition the textures into multiple compositions based on the atlas telemetry data, when the texture atlases are eventually packed. To prepare for partitioning, we create a graph of textures from the pairs in which each pair resembles an edge between two textures as nodes with a weight based on the draw call batch breaking frequency. A visualization of such a texture graph can be seen in Figure 2.1.

2.3 Prebuilt data-guided atlases

Using the atlas compositions and the texture graph we build data-guided texture atlases before run-time, usually during the phase in which the textures are also compressed if necessary. The data-guided atlases are built automatically without any manual intervention necessary by packing the textures according to the atlas compositions with a rectangle bin packing algorithm. When the rectangle bin packing algorithm cannot pack the textures into one atlas, the textures can be partitioned based on the texture graph or according to the bin packing algorithm itself. When using the texture graph, the composition is split up into multiple compositions by partitioning the texture graph with textures of the original composition with a graph partitioning algorithm. By partitioning the graph, it is split into multiple components that resemble the new texture atlas compositions. The textures in a single graph component need to fit into one atlas, which adds an additional constraint. The partitioning should also be balanced based on the weights for optimized draw call batching. If a partitioning is unbalanced, for example using a minimum cut algorithm, one texture at a time could be split off until we have multiple separate textures and one coherent fitting component. The leftover textures might not be related in any way, so this is not optimal nor a good approximation. In fact, an optimal solution for balanced graph partitioning is NP-complete [2], although in our case we do not need our components to be of equal size, which makes the problem space even larger. When the texture atlases have been created we store them on disk and we can run the game with textures in the prebuilt data-guided texture atlases.

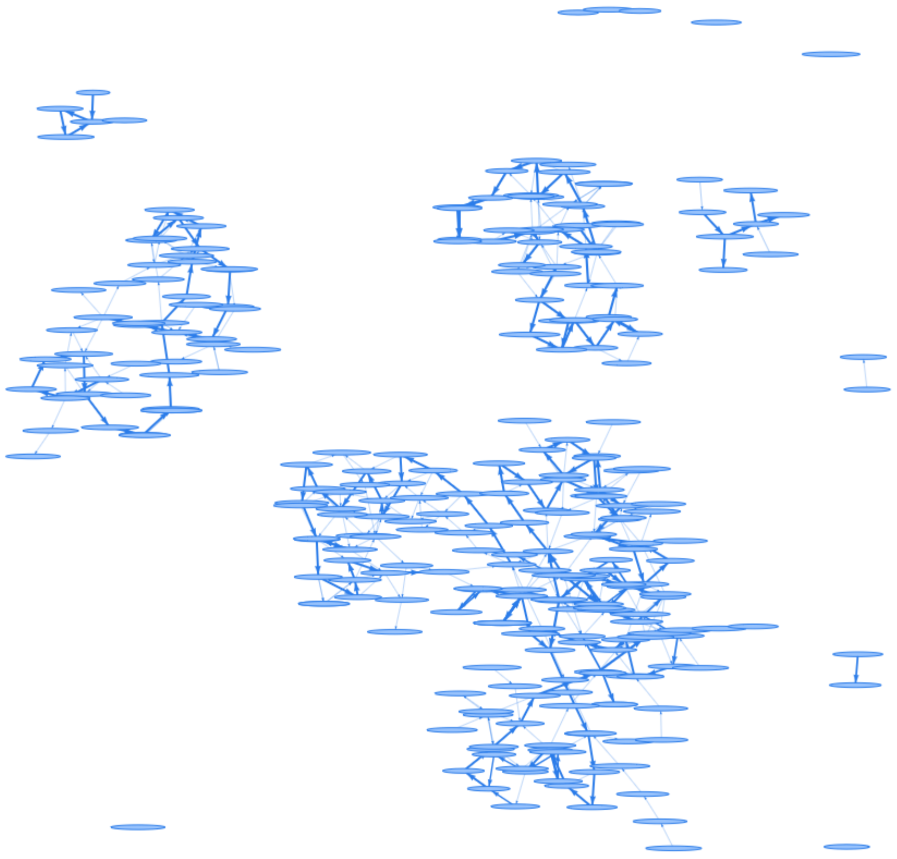


Figure 2.1: A visualization of a texture graph.

2.4 Transient atlases

Although we have automated the composition of texture atlases, the texture atlases are still not prepared for emergent and unpredictable use of textures as is the case in our running example introduced in Chapter 1, the assembly of a team of characters with different sets of textures. To accomplish this we create transient data-guided texture atlases. These atlases are transient, because they only exist in GPU memory while running a game, and might not even have to exist for the whole run. While running a game we gather which textures are loaded. After we have finished loading those textures into GPU memory, we create new atlases from the loaded textures. Loaded textures can be texture atlases as well as separate textures. We combine the loaded textures into a larger texture atlas composition by checking their mipmaps and format. Thereafter we pack the textures with rectangle bin packing and create a new texture in GPU memory that can contain the larger texture atlas composition. If the packing of the new texture atlas composition has larger dimensions than the maximum dimensions of a texture on the GPU, a partitioning algorithm is used to partition the transient texture atlas composition into multiple compositions based on the texture graph, as is also performed for prebuilt data-guided atlases. After valid compositions have been created, we copy each loaded texture to its transient texture atlas on the GPU, if it was put into a transient texture atlas composition. This operation needs to lead to a valid replacement of the loaded textures with the newly created transient atlases. When we create transient data-guided atlases for our running example, we can combine the characters of our assembled team into one transient texture atlas. If that does not fit we can partition the textures of the assembled team according to the texture graph. We expect a reduction of the number of draw calls for drawing textures of our assembled team by combining the textures into one or more transient atlases.

Chapter 3

Implementation

We implemented our approach, discussed in Chapter 2, in the Abbey Games game engine *AbbeyCore* and created a tool for receiving rendering data on a server and another for processing the rendering data and storing the output, so a game can use it to guide its texture atlas compositions.

3.1 AbbeyCore

AbbeyCore is a proprietary general purpose cross-platform 2D and 3D game engine developed in-house by Abbey Games written in C++. The engine supports scripting in Lua and hotloading scripts and assets. AbbeyCore targets the Direct3D and OpenGL graphics hardware APIs. We implemented our approach for Direct3D 11 in AbbeyCore. AbbeyCore 1.0 was used for Renowned Explorers: International Society[17]. Currently AbbeyCore is under further development and has passed version 2.0.

3.2 Atlas telemetry

The telemetry was implemented in the engine and gets notified of rendering events, such as the beginning of a new frame, the ending of a frame, when a new texture is bound and when a draw call is issued. The telemetry runs on a separate thread and collects the data associated with the events. When a frame begins, new frame data is allocated in which the data for the batches can be stored. Batch data consists of a list of texture data for one or more textures that were bound, because there are multiple texture slots available, and of a boolean that indicates whether the render pipeline has only changed due to texture binding or not. Texture data contains the name of the texture, the number of mipmap levels and the format. Each time a texture is bound, it is added to the current batch data and when a draw call is issued, the batch data is finished. Eventually the frame ends and then the frame data is finished. After a frame has ended, the telemetry performs preprocessing on the frame data to reduce the total amount of data that needs to be sent to the atlas telemetry server, because running a game at 60 frames per second with 1000 draw calls leads to a lot of data. When preprocessing a frame, the properties of data for a texture are hashed to avoid a lot of data duplication, because the same textures can be used in subsequent frames and even in other draw call batches in the same frame. For each pair of subsequent draw call batches we check if the pair could be batched together. This is true if only a binding of a different texture triggered the two draw calls and if the bound textures of the first and second draw call have the same number of mipmap levels and the same format. Keeping track of all texture paths through the draw calls that can be batched is too expensive, so we keep track of the frequency of the occurrence of pairs of textures in draw calls that can be batched[1]. The resulting data containing the properties of the textures and the frequency of their pairs is sent to the atlas telemetry server in a non-pretty-printed JSON format only when enough frames have been preprocessed to reduce memory usage, network bandwidth usage and load on the server.

The atlas telemetry server runs inside a Docker container and is situated locally on a server on the Abbey Games network. It was built using Node.js, TypeScript and Express using a PostgreSQL database. The server collects the data for running each game and keeps track of the period in which the data was sent, because development, patches, and expansions can change texture usage over time. We implemented weekly periods, but other periods of time can also be used depending on the developer's needs.

3.3 Atlas telemetry data processing

All the texture pairs of a game over all periods are used to create a texture graph, where a texture is a node and if two textures have a frequency as a pair, there is an edge between them with the frequency as the weight. The singular textures are added as a single node. Texture atlas compositions consist of a list of textures and are composed by computing the connected components[11] of the texture graph. The textures as nodes of each resulting connected component form an atlas composition. The contribution factor of the frequencies of the texture pairs are also diminished by half per period the further in the past the period was. We chose to diminish the contribution factor by half, other factors could also be used. When all the weights of the texture graph have been computed, the weights are normalized to a rational number between 0 and 1. The atlas compositions and the texture graph are written to the storage of the game, so the game can make use of the compositions and the texture graph. A visualization of the texture graph can be seen in Figure 2.1.

3.4 Prebuilt data-guided atlases

Source textures are stored in an uncompressed format and can be compressed during a build phase of the game. The build phase is executed before running the game and also before distribution of the game. In this phase texture atlases are also built. The uncompressed textures are read and packed based on the atlas compositions, previously composed manually. For prebuilt data-guided atlases the atlas compositions are composed by processing the atlas telemetry data. We use these atlas compositions to pack textures. Before packing, the textures in the atlas compositions are checked for consistency, because the format and the number of mipmap levels could have changed intermediately and could not be suitable for the composition anymore. If this is the case we exclude the texture from the composition.

3.5 Texture packing

Texture packing produces one or more texture atlas arrangements. An arrangement consists of the minimum width and height of the atlas and a texture region for each texture packed into the arrangement. A texture region has a source texture and a destination rectangle in the texture atlas. When finally building the texture atlas the texture is copied from the source texture to the destination rectangle in the texture atlas for each mipmap level. Two offline rectangle bin packing algorithms were implemented for packing textures into atlas arrangements based on compositions. Offline algorithms were used, because the atlas compositions with the input textures were known before texture packing. The time and space complexities of the two algorithms seemed feasible for our approach, while the algorithms also perform an almost optimal packing[14].

The first one was a shelf first fit bin first fit decreasing height algorithm (SHELF-FF-BFF-DESCH) for which the input textures were sorted based on decreasing height before packing and a texture is placed on the first level that fits. If a texture does not fit on any level, a new texture atlas arrangement or *bin* is created and the texture is placed in the new bin. The next texture is placed in the first bin that fits[14].

The second one was a maximum rectangles best shortest side fit bin first fit decreasing shortest side algorithm (MAXRECTS-BSSF-BFF-DESCSS) for which the input textures were sorted based on decreasing shortest side before packing and a texture is placed in the rectangle with the best shortest side fit. If a texture does not fit into any rectangle, a new bin is created and the texture is placed in the new bin. The next texture is also placed in the first bin that fits[14].

3.6 Alignment

To avoid texture bleeding on lower mipmap levels the textures are padded or aligned on a square grid, depending on the rectangle bin packing algorithm. For

MAXRECTS the textures were padded with the difference between the size of the texture and the texture size plus the size of a grid square aligned to the size of a grid square. For FFDH the textures coordinates were aligned on the grid squares and at least one grid square is between each texture. The formula for the size of a grid square is

$$\gamma = \max\{2^{m-1}, b\}$$

for prebuilt texture atlases and

$$\gamma = 2^{m-1} * b$$

for transient texture atlases, where γ is the square grid size, m is the number of mipmap levels and b is the block size of the format. Usually for compressed textures b is 4 pixels and for uncompressed textures it is 1 pixel. For prebuilt texture atlases the grid does not have to be aligned to the block size, because the textures to be packed are still uncompressed. For transient texture atlases however, the target texture atlas for compressed packed textures is already compressed, so to be able to directly copy a texture into the transient texture atlas, we need align the grid to the compression block size.

The dimensions of a texture atlas also need to be aligned for consistent texture coordinates when using mipmaps and compression. The formula for computing the alignment necessary is

$$\alpha = 2^{m-1} * b$$

where α is the alignment, m is the number of mipmap levels and b is the block size of the format. The formula for computing the right dimensions for a texture is

$$\delta = \min\{\lceil s/\alpha \rceil * \alpha, 2^{\lceil \log_2 s \rceil}\}$$

where δ is a dimension of the texture, s is the original dimension of the texture, which for a 2D texture can either be the width or the height, and α is the alignment. A dimension is either aligned or ceiled to the nearest power of two, which also works.

3.7 Texture partitioning

By default the chosen rectangle bin packing algorithm puts a texture into a new atlas when the current atlas is full, so textures are partitioned based on the order of the input textures. This means that textures are not partitioned based on batch breaking, but on a heuristic on a property of a single texture. Instead of relying on a heuristic that may or may probably not produce a partitioning that reduces draw calls, we can use the texture graph to partition textures into atlases based on texture pair batch breaking frequency. When the texture packer cannot fit the textures of the atlas composition into one atlas, a partitioning algorithm is used. We developed two partitioning algorithms in addition to the default of the rectangle bin packing algorithm. A rectangle bin packing algorithm still takes care of the arrangement of the atlases, a partitioning algorithm only takes care of partitioning the composition that did not fit into a single atlas arrangement.

The greedy algorithm packs the textures in the texture graph that are most frequently sequentially drawn into one atlas, so that at least those are together in an atlas. First the edges in the texture graph are sorted based on their weight in decreasing order. Thereafter the first edge is dequeued and the two textures at the end of both sides are packed into one atlas, if not already packed. When a packing overflows into multiple atlases, the previous packing is used as it only had one atlas and the resulting atlas is added to the collection of partitioned atlases. We repeat dequeuing edges until there are no edges left.

The approximation algorithm partitions the textures in the texture graph into the most tightly connected components using a Stoer-Wagner-like algorithm[19]. First the textures in the composition are sorted based on the sum of the edge weights of neighboring textures in decreasing order. The texture that is the most tightly connected to its neighbors is picked and added to the tightly connected textures composition. From there the tightly connected textures composition is grown by adding the neighboring texture, that is the most tightly connected to the all the textures in the composition by summing the weights to the textures in the composition, until the composition overflows. When the composition overflows, the previous composition is stored and it starts over by growing another tightly connected textures composition, until all textures are partitioned and packed.

Offline nearly optimal rectangle bin packing algorithms were used for partitioning to approximate optimal packing. However, online suboptimal algorithms have a better time complexity, but worse space usage and might have been more suitable for partitioning or checking whether or not a composition fits. Especially for transient atlases the time complexity could become problematic when partitioning a large amount of textures.

3.8 Transient atlases

For creating transient texture atlases the engine keeps track of the sprites and their corresponding textures that are loaded for the game. When all textures have been loaded, the transient atlases are composed and arranged on a separate thread, so the work can be spread over multiple frames if necessary, without decreasing the frame rate of the game. Composing and arranging transient atlases is done by executing Algorithm 1. Afterwards the transient texture atlas arrangements are sent over to the main thread and then commands are given to the render thread which communicates with the GPU to actually create the transient texture atlases in Algorithm 2. The number of in-GPU-memory texture region copies per frame can be limited to ensure an interactive frame rate, but possibly spreads the process of creating transient texture atlases over multiple frames, so multiple texture atlases with unnecessary textures could stay in GPU memory and increase memory usage. We did not limit the number of in-GPU-memory texture region copies per frame for our experiments in Chapter 4.

Algorithm 1: The composition and arrangement of transient atlases.

```
Data: $S$ = loaded sprites, $G$ = texture graph, $packer$, $partitioning$
Result: Transient texture atlas arrangements
$C = \{\}$;
for $sprite \in S$ do
    // Can only pack a texture if dimensions are aligned
    if $aligned(sprite.texture)$ then
        // Check if sprite texture is already in a composition
        if $C.HasCompositionWith(sprite.texture)\}$ then
            $C.GetCompositionWith(sprite.texture).S.insert(sprite)$;
        else
            // Check if there is already a composition with
            // the same number of mipmap levels and format,
            // if not create it
            $mipLevels = sprite.texture.mipmapLevels$;
            $format = sprite.texture.format$;
            if $C.HasCompositionFor(mipLevels, format)$ then
                $composition =$
                  $C.GetCompositionFor(mipLevels, format)$;
            else
                $composition = new\ Composition(mipLevels, format)$;
                $C.insert(composition)$;
            end
            // Keep track of the sprites using the texture
            $composition[sprite.texture].S = \{sprite\}$;
        end
    end
end
// Create transient texture atlas arrangements
$A = \{\}$;
for $composition \in C$ do
    $A.insert(packer.packTextures(composition, partitioning, G))$;
end
return $A$
```

Algorithm 2: The creation of transient texture atlases.

```
Data: $A$ = transient texture atlas arrangements
for $a \in A$ do
    // Create a transient texture atlas in GPU memory
    $texture =$
     $new\ Texture2D(a.width, a.height, a.mipmapLevels, a.format)$;
    $blockSize = GetBlockSize(a.format)$;
    for $region \in a.regions$ do
        // The current width of the mipmap level
        $width = region.width$;
        // The current height of the mipmap level
        $height = region.height$;
        // The source rectangle
        $source = (0, 0, region.width, region.height)$;
        // The destination position in the atlas
        $destination = (region.x, region.y)$;
        for $mipLevel \leftarrow 0$ to $a.mipmapLevels$ do
            // Copy the region of a mipmap of a texture to
            // the transient texture atlas in GPU memory
            $region.texture.CopyTo(source, mipLevel, texture, destination)$;

            // Divide by two for the next mipmap level
            $width = \max\{width\ /\ 2, blockSize\}$;
            $height = \max\{height\ /\ 2, blockSize\}$;
            $source.x = source.x\ /\ 2$;
            $source.y = source.y\ /\ 2$;
            $source.z = source.x + width$;
            $source.w = source.y + height$;
            $destination.x = destination.x\ /\ 2$;
            $destination.y = destination.y\ /\ 2$;
        end
        // A region can be a texture atlas,
        // so we need to point all sprites to
        // their place in the new texture
        for $sprite \in region.S$ do
            $sprite.UpdateTextureCoordinates(region.u, region.v, a.width, a.height)$;

            $sprite.SetTexture(texture)$;
        end
    end
end
```

Chapter 4

Experimental Setup

In this chapter we explain the setup and design of a set of experiments and discuss the usage of the approach proposed in Chapter 2 in these experiments.

4.1 Simulations

We gathered results from multiple simulations to show what strengths and weaknesses the proposed approach entails and how varying and combining different algorithms and parameters for the approach perform in comparison. The camera does not move during all simulations and is positioned directly in front of the textures with all textures facing the camera plane with their normals. There is no user interaction required for the simulation to make the simulations as reproducible and stable as possible.

The first simulation is a real world scenario in which a dynamic 2D scene is simulated. The scene contains multiple sprites and animations with many textures. The scene is suitable for an experiment, because it is typical for an Abbey Games game and was created by the Abbey Games artists and developers. We can measure how our approach performs in the real world with this simulation.

The second simulation is an artificially composed scene with a set of static textures that serves as a baseline. The textures are always drawn the same way each frame, so we can measure the effectiveness of data-guided atlases and the overhead and impact on performance of creating and using transient atlases during the whole simulation.

The third simulation is an artificially composed scene for which combinations of groups of static textures are used. Each time the simulation is run five groups of static textures are loaded and shown at random out of a total of ten groups. This simulates our running example introduced in Chapter 1. The simulation was created to test the effectiveness of transient atlases, because our approach should improve draw call batching by combining the randomly chosen groups of textures into a transient atlas at run-time. A frame of the simulation can be

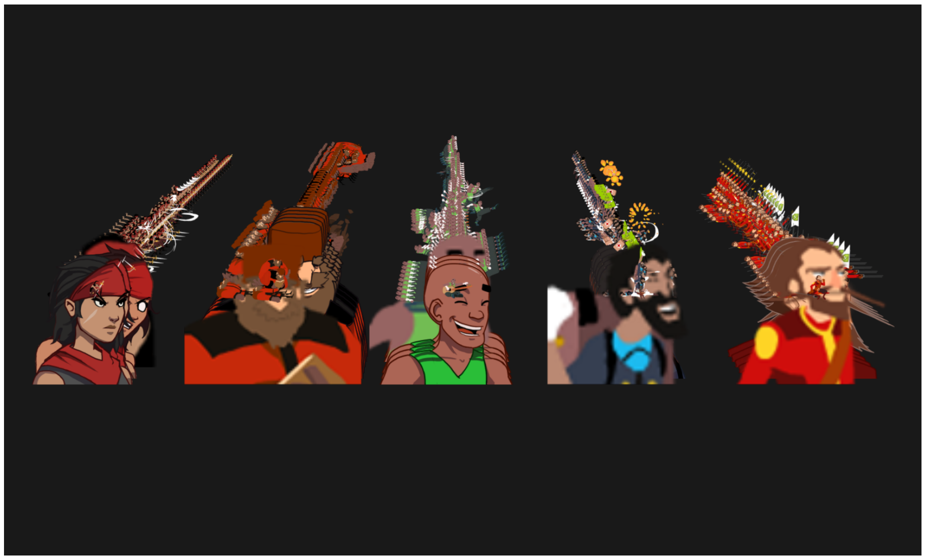


Figure 4.1: A frame of simulation 3 where five characters have been selected.

seen in Figure 4.1.

The fourth simulation is an artificially composed scene with sets of textures of which each set dynamically changes position in a random x- and z-direction in order to trigger overlap, occlusion and draw order changes. This simulation was created to check if our approach also improves performance for coherently moving groups of textures.

Other simulations could have been formulated. For example a simulation where textures dynamically change positions randomly. However, simulations usually have some form of visual coherence with respect to their textures in order to provide a meaningful perception[27].

The textures used in the second, third and fourth simulation were taken from the game Renowned Explorers: International Society[17]. An encounter in the game can be seen in Figure 4.2. In the encounter a crew of characters, as can be seen in the middle of Figure 4.2, duels versus opponents. The player can select the characters for the crew, which perfectly resembles our running example.

4.2 Setup

Simulations were written in Lua and were run on AbbeyCore using the implementation discussed in Chapter 3. In Table 4.1 specifications of the system used for performing the experiments are specified.

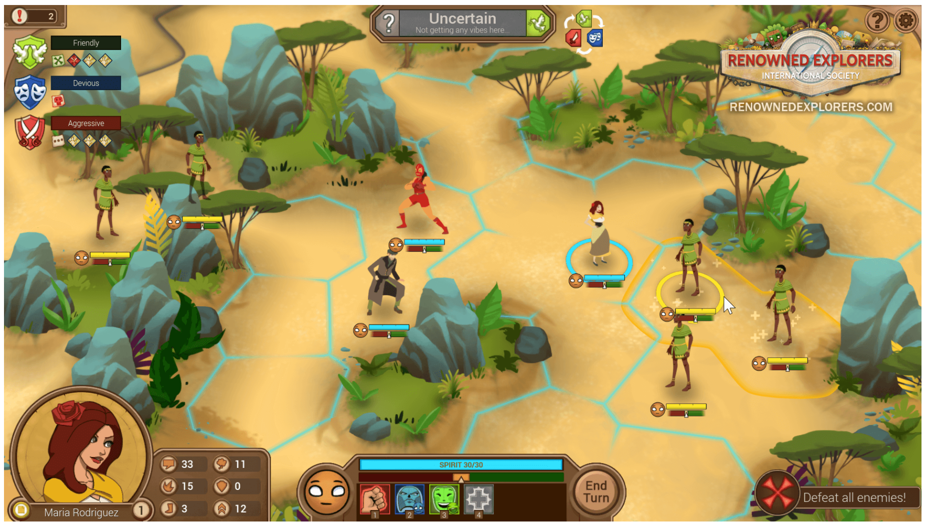


Figure 4.2: An encounter in Renowned Explorers: International Society[17].

Table 4.1: System specifications

OS	Microsoft Windows 10 Home 10.0.15063 Build 15063
Motherboard	Intel HM175
CPU	Intel Core i7-7700HQ CPU @ 2.80GHz, 4 Cores, 8 Logical Processors
GPU	NVIDIA GeForce GTX 1070 Mobile - 8192 MB, Core: 1443 MHz, Memory: 4000 MHz, GDDR5, 256-bit interface, no Optimus
RAM	16384 MB, DDR4-2400, single-channel
Harddisk 1	Samsung SSD PM871a MZNLN256HMHQ, 256 GB
Harddisk 2	HTS721010A9E630, 1 TB HDD, 7200 rpm
Display	15.6 inch 16:9, 1920x1080 pixel 141 PPI, LG Philips LP156WF6 (LGD046F), IPS, G-Sync, 60 Hz, Full HD

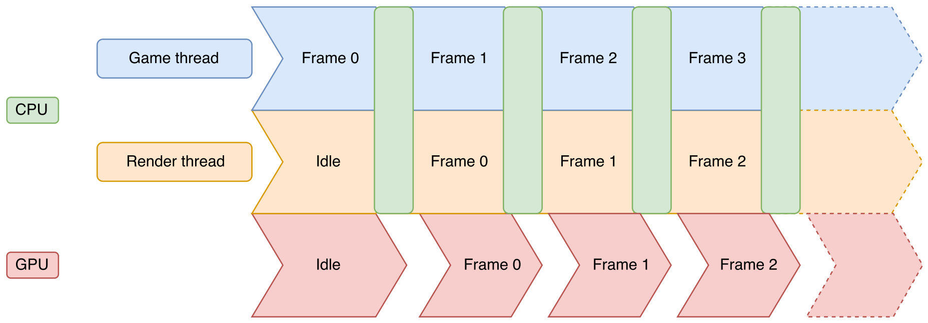


Figure 4.3: A high-level overview of the CPU and GPU usage of AbbeyCore including the game and render thread executing on the CPU. A green rectangle overlapping the game and render thread flows represents a synchronization point.

4.3 Performance metrics

We measured performance directly in the engine using numerous metrics.

For measuring how well our packing and partitioning algorithms performed we tracked the CPU time it took to partition and pack an atlas in milliseconds, the used area of the atlas in square texels, the total area of the atlas in square texels, the occupancy of the atlas as a percentage of the used space of the atlas and the number of textures per atlas.

For our simulations we measured the time spent in the render thread per frame on the CPU in milliseconds, the time spent per frame on the GPU in milliseconds, the number of draw calls and the average number of primitives per draw call per frame. Performance was measured from the moment all textures are loaded, but before the moment that initiates the creation of transient atlases from the loaded textures, because we want to measure the overhead and the impact of creating and using transient texture atlases. AbbeyCore uses a multithreaded setup on the CPU including a game thread and a render thread. An overview of this setup can be seen in Figure 4.3. The game thread executes the scripting code and the render thread on the CPU keeps track of GPU resources and communicates with the GPU. The render thread renders the game for the previous frame while the game thread prepares the next frame for rendering. Both threads have a synchronization point in common after rendering the previous frame and after the game has prepared the next frame to prevent one another from continuing while the other has not finished yet. Only the render thread communicates with the GPU, which means that any CPU bottleneck due to draw calls is to be found on the render thread.

The overhead when running a game on the engine with atlas telemetry enabled is rather significant, but not disturbing as it runs on a separate thread. However, at Abbey Games atlas telemetry is only be enabled during development, playtesting and QA testing, so it does not impact the end user experience.

Data was also sent over a local network for reduced memory overhead, which grows when waiting for a request due to the sequential nature of the telemetry requests. The performance was assessed without enabling atlas telemetry to simulate the end user experience.

Instrumentation overhead for measuring the performance metrics was either negligible or had no impact on the measurements itself. The instrumentation was implemented directly in the relevant parts of the engine for accurate measurements and to avoid impact by performing resource-intensive tasks out of scope of the measurements.

4.4 Parameter configurations

The parameters that have an effect on building texture atlases when building a simulation are the maximum texture size, the texture packing algorithm and the partitioning algorithm for partitioning textures into multiple atlases for an overflowing texture atlas. The maximum texture size should always be conservative, because some older hardware APIs that should be able to run the game, might not be able to use larger textures that do run on newer hardware APIs. For example Direct3D 11 supports 16384 texels in both dimensions. AbbeyCore also targets Direct3D 10 in addition to Direct3D 11, which supports 2D textures with up to 8192 texels in both dimensions, so when building a simulation we cannot create larger textures than 8192 by 8192 texels. The maximum texture size can also be set lower than the hardware API capability, for example to 4096 texels and might benefit transient atlas compositions or memory usage when only some textures in a texture atlas are used. The texture packing algorithm can be either the FFDH or the MAXRECTS bin packing algorithm[14]. The partitioning algorithm for overflowing atlases can be the default of the chosen rectangle bin packing algorithm, the greedy algorithm or the approximation algorithm.

When running a simulation and loading the textures used for a scene we also have to choose whether transient atlases are enabled or not and if so what texture packing and what texture partitioning algorithm are used. The maximum texture size is not configurable for a transient texture atlas, because a larger texture atlas can be composed of more textures when necessary, so performance is improved when that leads to less draw calls and the performance gain outweighs the small memory overhead of a transient texture atlas. When atlas transience is enabled textures are packed with a maximum texture size of the hardware API of the GPU. In our case this was 16384 texels in both dimensions.

An overview of all these parameters and their possible values can be seen in Table 4.2.

To avoid combinatorial explosion and data irrelevance some parameter configurations were not executed. The texture packing and partitioning algorithms were not varied with different combinations for building texture atlases when building a simulation and packing transient atlases. If we did vary those algorithms we would have needed much more time for executing all the parameter configurations, because this would multiply the number of different possible configurations by at least six. Running a simulation with prebuilt separate textures or with prebuilt separate textures with transient texture atlases enabled made varying some parameters irrelevant for execution, such as the maximum texture size, so we only performed executions with parameter configurations with varying parameters that had an effect on the execution itself. Full details on which configurations were run for which simulation are shown in the results in Chapter 5.

Table 4.2: Configuration parameters

Packing algorithm	FFDH	MAXRECTS	
Partitioning algorithm	Default	Greedy	Approximately
Maximum texture size	4096	8192	
Atlas transience	Enabled	Disabled	

4.5 Execution

For each simulation we executed three phases. The first phase was executed only once for each simulation and the second and third phases were executed for each parameter configuration for each simulation.

The first phase is a data collection and analysis phase in which we enabled atlas telemetry for the engine and sent data to the atlas telemetry server while running a simulation a hundred times with separate textures, so no texture atlases, for 300 frames. The simulation was run for a hundred times to account for the randomness in the first, third and fourth simulations. After the first phase we have gathered enough data for analysis and we output the input atlas compositions and the texture graph for AbbeyCore.

In the second phase we built the simulation with the engine. This phase was also varied in three ways in addition to the parameter configurations. As building the simulation includes building the textures and texture atlases, intuitively there are two choices: build separate textures or build texture atlases. However, for three out of the four simulations we had original input atlas compositions manually created by the developers of Renowned Explorers: International Society[17]. For those simulations the manual atlas compositions were also used, because these were unbiased and the use of the textures was similar to their use. So there were three ways of building the textures: separately, texture atlases built using the input atlas compositions created from the atlas telemetry data in the first phase and texture atlases using the manual atlas compositions.

After build completion the simulation was run twenty times for a limited number of frames in the third phase. The number of frames a simulation runs was limited to 600 to get a good stable profile sample of the performance of a simulation. The simulation was run multiple times to account for profiling sample outliers due to randomness or to detect other anomalies. Randomness is expected to not affect performance randomly, because it determined only the use of different textures in the first and third simulation and no additional events other than overlap, occlusion and draw order changes in the fourth simulation.

Chapter 5

Results

In this chapter we show the results of the experiments conducted according to the setup in Chapter 4.

5.1 Overall performance

For each simulation, detailed in Section 4.1, the overall performance was analyzed and the averages of each performance measurement were put into Table 5.1, 5.2, 5.3 and 5.4. Each of those tables shows what was prebuilt in the topmost row. The second row shows if the simulation was run with transience enabled or just with what was prebuilt. When a simulation is run with transience what was prebuilt is also used. The results are averages of the twenty executions of 600 frames of each parameter configuration with the exception of the transient parameter.

Table 5.1: Average performance measurements for all configurations ran with simulation 1.

	Data-Guided Atlases		Separate Textures	
Measurement	Prebuilt	Transient	Prebuilt	Transient
CPU Frame Time (ms)	6.78	6.76	11.44	7.92
GPU Frame Time (ms)	7.64	7.65	12.24	8.73
#Draw Calls	345.10	343.26	1098.71	363.89
#Primitives per Draw Call	21.89	21.91	8.86	26.33

In Table 5.1 the manual atlases measurements are missing, because there were no manually created atlases available for real-world simulation 1. We observe that on average the performance for non-transient and transient data-guided atlases was very similar for the real world scenario in simulation 1 and that separate textures without transient atlases performed the worst, as expected. On average the performance of prebuilt separate textures with transient atlases is better than completely separate textures and is quite close to the performance of data-guided atlases, especially regarding the number of draw calls.

Table 5.2: Average performance measurements for all configurations ran with simulation 2.

	Data-Guided Atlases		Separate Textures		Manual Atlases	
Measurement	Prebuilt	Transient	Prebuilt	Transient	Prebuilt	Transient
CPU Frame Time (ms)	3.82	2.40	10.20	2.97	7.24	4.07
GPU Frame Time (ms)	4.02	2.59	10.43	3.18	7.46	4.28
#Draw Calls	338.07	5.91	1608.89	84.57	1090.92	318.23
#Primitives per Draw Call	551.09	765.85	2.10	1042.65	3.11	14.56

For static textures simulation 2 the results were a lot less similar. In Table 5.2 the transient data-guided atlases performed the best on average, although the prebuilt separate textures with transient atlases were also quite performant. The manual atlases performed badly without transient atlases and even with transient atlases enabled it could not match transient data-guided atlases nor transient atlases for separate textures. We observe that the number of draw calls had a severe impact on performance of both the CPU and GPU when only rendering static geometry with a lot of textures.

The number of draw calls and the average number of primitives per draw call are inversely proportional and the proportion should roughly be the same for each execution of the same simulation regardless of parameter configuration, but in the prebuilt data-guided atlases column in Table 5.2 the number of draw calls and the average number of primitives per draw call are both very high compared to the rest. This also occurs in some columns in Table 5.3 and 5.4 and was caused by averaging highly varying performance measurements of the configurations. We elaborate on the varying performance of the parameter configurations in the next section and we also expect to see the stability and inverse proportionality of the number of draw calls and the average number of primitives per draw call.

Table 5.3: Average performance measurements for all configurations ran with simulation 3.

	Data-Guided Atlases		Separate Textures		Manual Atlases	
Measurement	Prebuilt	Transient	Prebuilt	Transient	Prebuilt	Transient
CPU Frame Time (ms)	4.57	2.43	9.25	2.66	3.45	2.52
GPU Frame Time (ms)	4.77	2.62	9.47	2.86	3.64	2.71
#Draw Calls	542.33	44.01	1436.21	72.31	263.22	51.52
#Primitives per Draw Call	7.95	75.58	2.11	94.83	11.34	58.88

Our running example in simulation 3, Table 5.3, shows that on average the prebuilt manual atlases outperformed the prebuilt data-guided atlases, because of the unpredictability. However, transient data-guided atlases slightly outperformed transient manual atlases. We observe that transient atlases were essential for optimal performance when dealing with team assembly.

Table 5.4: Average performance measurements for all configurations ran with simulation 4.

	Data-Guided Atlases		Separate Textures		Manual Atlases	
Measurement	Prebuilt	Transient	Prebuilt	Transient	Prebuilt	Transient
CPU Frame Time (ms)	6.16	4.60	12.11	5.29	9.60	5.95
GPU Frame Time (ms)	12.06	11.82	12.79	12.55	12.67	12.51
#Draw Calls	331.95	7.25	1533.81	64.60	1028.22	212.37
#Primitives per Draw Call	552.17	767.63	2.02	1056.65	3.26	15.80

In Table 5.4, for simulation 4 with the static coherently moving textures, the GPU frame time was rather high compared to the GPU frame time for other simulations independent of the number of draw calls. The CPU frame time however was heavily influenced by the number of draw calls. We also see that the decrease of CPU frame time due to the reduction of draw calls diminishes when the number of draw calls has become lower and lower.

5.2 Configuration performance

In Table 5.5, 5.6, 5.7 and 5.8 the average performance results per parameter configuration and per performance measurement are shown. Each performance measurement has a minimum or maximum value in bold. We want to minimize CPU frame time, GPU frame time and the number of draw calls, so for these columns the minimum is emboldened. For the average number of primitives per draw call the opposite is the case, so the maximum has been emboldened. The configurations are listed according to the following format: the type of prebuilt textures (separate textures, data-guided atlases or manual atlases), the maximum texture size for prebuilt atlases, which is not relevant and left out for separate textures, the texture packing algorithm and the partitioning algorithm. The transience is encoded as the highlight of the row, if a row is highlighted, transient atlases were enabled.

Table 5.5: Average performance measurements of simulation 1 for each configuration. Highlighted rows indicate that transience was enabled.

Configuration	CPU Frame Time (ms)	GPU Frame Time (ms)	#Draw Calls	#Primitives per Draw Call
Separate Textures	11.44	12.24	1098.71	8.86
Separate Textures FFDH Approx.	7.91	8.70	362.87	26.38
Separate Textures FFDH Default	7.95	8.78	364.21	**26.43**
Separate Textures FFDH Greedy	7.91	8.73	363.12	26.37
Separate Textures MAXRECTS Approx.	7.90	8.68	364.66	26.18
Separate Textures MAXRECTS Default	7.93	8.74	364.01	26.34
Separate Textures MAXRECTS Greedy	7.92	8.73	364.44	26.29
Data-Guided Atlases 4096 FFDH Approx.	6.88	7.72	347.74	22.11
Data-Guided Atlases 4096 FFDH Approx.	6.88	7.75	349.26	22.20
Data-Guided Atlases 4096 FFDH Default	6.91	7.77	347.97	22.11
Data-Guided Atlases 4096 FFDH Default	6.97	7.81	349.42	22.16
Data-Guided Atlases 4096 FFDH Greedy	6.88	7.75	349.15	22.03
Data-Guided Atlases 4096 FFDH Greedy	6.96	7.83	349.84	22.08
Data-Guided Atlases 4096 MAXRECTS Approx.	6.75	7.65	**328.22**	21.54
Data-Guided Atlases 4096 MAXRECTS Approx.	6.51	7.41	328.72	21.57
Data-Guided Atlases 4096 MAXRECTS Default	6.50	7.35	346.44	20.51
Data-Guided Atlases 4096 MAXRECTS Default	6.47	7.36	332.54	21.38
Data-Guided Atlases 4096 MAXRECTS Greedy	**6.42**	**7.30**	329.41	21.48
Data-Guided Atlases 4096 MAXRECTS Greedy	6.45	7.31	329.51	21.60
Data-Guided Atlases 8192 FFDH Approx.	6.77	7.61	347.96	22.16
Data-Guided Atlases 8192 FFDH Approx.	6.81	7.63	349.95	22.25
Data-Guided Atlases 8192 FFDH Default	6.87	7.74	348.37	22.13
Data-Guided Atlases 8192 FFDH Default	6.94	7.80	348.06	22.13
Data-Guided Atlases 8192 FFDH Greedy	6.88	7.71	348.28	22.19
Data-Guided Atlases 8192 FFDH Greedy	6.92	7.77	347.85	22.14
Data-Guided Atlases 8192 MAXRECTS Approx.	6.95	7.83	349.83	22.18
Data-Guided Atlases 8192 MAXRECTS Approx.	6.61	7.80	333.46	21.05
Data-Guided Atlases 8192 MAXRECTS Default	6.77	7.60	349.28	22.03
Data-Guided Atlases 8192 MAXRECTS Default	6.82	7.67	350.23	22.20
Data-Guided Atlases 8192 MAXRECTS Greedy	6.78	7.65	348.49	22.16
Data-Guided Atlases 8192 MAXRECTS Greedy	6.82	7.68	350.33	22.10

The real world scenario of simulation 1 with results for each configuration in Table 5.5 performed similar for each configuration with separate textures as well as with data-guided atlases. A partitioning of textures into smaller texture atlases of 4096 by 4096 texels was apparently slightly better than a partitioning into larger texture atlases for data-guided atlases. All configurations of separate textures with transient atlases performed worse than all configurations of the data-guided atlases.

Table 5.6: Average performance measurements of simulation 2 for each configuration. Highlighted rows indicate that transience was enabled.

Configuration	CPU Frame Time (ms)	GPU Frame Time (ms)	#Draw Calls	#Primitives per Draw Call
Separate Textures	10.20	10.43	1608.89	2.10
Separate Textures FFDH Approx.	2.58	2.79	5.58	1095.55
Separate Textures FFDH Default	2.57	2.77	4.36	1096.16
Separate Textures FFDH Greedy	2.57	2.78	5.04	1095.54
Separate Textures MAXRECTS Approx.	3.38	3.59	168.61	986.40
Separate Textures MAXRECTS Default	3.37	3.57	164.95	989.03
Separate Textures MAXRECTS Greedy	3.34	3.56	158.86	993.20
Data-Guided Atlases 4096 FFDH Approx.	4.76	4.96	555.22	5.97
Data-Guided Atlases 4096 FFDH Approx.	2.43	2.62	8.61	468.84
Data-Guided Atlases 4096 FFDH Default	4.65	4.85	526.36	6.29
Data-Guided Atlases 4096 FFDH Default	2.42	2.60	7.78	469.49
Data-Guided Atlases 4096 FFDH Greedy	5.09	5.30	645.76	5.14
Data-Guided Atlases 4096 FFDH Greedy	2.40	2.59	7.53	469.83
Data-Guided Atlases 4096 MAXRECTS Approx.	5.64	5.83	732.32	4.54
Data-Guided Atlases 4096 MAXRECTS Approx.	2.44	2.63	10.14	365.18
Data-Guided Atlases 4096 MAXRECTS Default	5.07	5.27	624.86	5.31
Data-Guided Atlases 4096 MAXRECTS Default	2.41	2.61	7.41	469.91
Data-Guided Atlases 4096 MAXRECTS Greedy	6.35	6.57	954.20	3.50
Data-Guided Atlases 4096 MAXRECTS Greedy	2.43	2.62	11.45	364.79
Data-Guided Atlases 8192 FFDH Approx.	**2.37**	**2.56**	**3.01**	1097.05
Data-Guided Atlases 8192 FFDH Approx.	2.38	2.57	**3.01**	1096.98
Data-Guided Atlases 8192 FFDH Default	2.38	**2.56**	**3.01**	1097.06
Data-Guided Atlases 8192 FFDH Default	2.39	2.57	**3.01**	1097.06
Data-Guided Atlases 8192 FFDH Greedy	2.38	2.57	**3.01**	1097.09
Data-Guided Atlases 8192 FFDH Greedy	2.38	2.57	**3.01**	1097.02
Data-Guided Atlases 8192 MAXRECTS Approx.	2.38	2.57	**3.01**	1096.93
Data-Guided Atlases 8192 MAXRECTS Approx.	2.39	2.57	**3.01**	1097.01
Data-Guided Atlases 8192 MAXRECTS Default	2.40	2.58	**3.01**	**1097.10**
Data-Guided Atlases 8192 MAXRECTS Default	2.39	2.58	**3.01**	1097.06
Data-Guided Atlases 8192 MAXRECTS Greedy	2.38	2.57	**3.01**	1097.05
Data-Guided Atlases 8192 MAXRECTS Greedy	2.38	2.58	**3.01**	1097.04
Manual Atlases 4096 FFDH Approx.	6.99	7.20	1044.74	3.20
Manual Atlases 4096 FFDH Approx.	3.53	3.73	209.95	15.73
Manual Atlases 4096 FFDH Default	7.13	7.34	1044.74	3.20
Manual Atlases 4096 FFDH Default	3.54	3.75	209.53	15.74
Manual Atlases 4096 FFDH Greedy	6.94	7.15	1044.74	3.20
Manual Atlases 4096 FFDH Greedy	3.53	3.74	209.74	15.74
Manual Atlases 4096 MAXRECTS Approx.	6.99	7.21	1046.73	3.20
Manual Atlases 4096 MAXRECTS Approx.	3.56	3.77	211.45	15.59
Manual Atlases 4096 MAXRECTS Default	6.95	7.17	1045.73	3.20
Manual Atlases 4096 MAXRECTS Default	3.55	3.76	211.03	15.60
Manual Atlases 4096 MAXRECTS Greedy	6.95	7.17	1045.73	3.20
Manual Atlases 4096 MAXRECTS Greedy	3.58	3.79	212.01	15.59
Manual Atlases 8192 FFDH Approx.	6.94	7.16	1044.74	3.20
Manual Atlases 8192 FFDH Approx.	3.53	3.73	209.32	15.74
Manual Atlases 8192 FFDH Default	6.98	7.20	1044.74	3.20
Manual Atlases 8192 FFDH Default	3.53	3.74	208.97	15.75
Manual Atlases 8192 FFDH Greedy	7.06	7.28	1044.74	3.20
Manual Atlases 8192 FFDH Greedy	3.54	3.74	209.39	15.74
Manual Atlases 8192 MAXRECTS Approx.	6.98	7.20	1046.73	3.20
Manual Atlases 8192 MAXRECTS Approx.	3.57	3.78	211.10	15.60
Manual Atlases 8192 MAXRECTS Default	6.95	7.17	1045.73	3.20
Manual Atlases 8192 MAXRECTS Default	3.56	3.77	211.73	15.59
Manual Atlases 8192 MAXRECTS Greedy	10.05	10.28	1591.97	2.13
Manual Atlases 8192 MAXRECTS Greedy	9.78	10.01	1504.58	2.25

For simulation 2 in Table 5.6 the results are a lot less decisive. Separate textures with transient atlases and the FFDH packing algorithm performed similar to data-guided atlases and consistently a lot better than the other packing algorithm for separate textures with transient atlases, MAXRECTS. Data-guided atlases configurations still consistently performed better than their manual atlases counterparts, but when decreasing the maximum texture atlas size for data-guided atlases performance went down and only transience could get it up to speed with the configurations of larger maximum texture size for data-guided atlases, which all performed equally well. The equality in performance is due to the fact that almost all textures fit into the texture atlas of 8192 by 8192 texels. The manual atlases with a maximum texture size of 8192, MAXRECTS packing algorithm and greedy partitioning made a rather large slip in performance, almost as bad as completely separate textures, which is rather odd compared to the other manual atlases configurations performing consistently.

Table 5.7: Average performance measurements of simulation 3 for each configuration. Highlighted rows indicate that transience was enabled.

Configuration	CPU Frame Time (ms)	GPU Frame Time (ms)	#Draw Calls	#Primitives per Draw Call
Separate Textures	9.25	9.47	1436.21	2.11
Separate Textures FFDH Approx.	2.41	2.61	31.49	100.63
Separate Textures FFDH Default	2.39	2.59	**31.42**	**102.50**
Separate Textures FFDH Greedy	2.46	2.66	32.56	97.67
Separate Textures MAXRECTS Approx.	2.98	3.19	122.08	85.27
Separate Textures MAXRECTS Default	2.90	3.10	112.82	91.71
Separate Textures MAXRECTS Greedy	2.82	3.03	103.49	91.23
Data-Guided Atlases 4096 FFDH Approx.	4.48	4.68	512.13	5.92
Data-Guided Atlases 4096 FFDH Approx.	2.46	2.66	47.75	69.35
Data-Guided Atlases 4096 FFDH Default	6.62	6.83	1012.65	3.05
Data-Guided Atlases 4096 FFDH Default	2.44	2.64	47.15	72.94
Data-Guided Atlases 4096 FFDH Greedy	4.28	4.48	473.38	6.41
Data-Guided Atlases 4096 FFDH Greedy	2.45	2.64	46.04	71.05
Data-Guided Atlases 4096 MAXRECTS Approx.	4.52	4.72	531.24	5.58
Data-Guided Atlases 4096 MAXRECTS Approx.	2.48	2.68	53.03	62.41
Data-Guided Atlases 4096 MAXRECTS Default	6.93	7.13	1122.84	2.67
Data-Guided Atlases 4096 MAXRECTS Default	2.43	2.62	47.04	70.18
Data-Guided Atlases 4096 MAXRECTS Greedy	4.89	5.09	622.53	4.83
Data-Guided Atlases 4096 MAXRECTS Greedy	2.45	2.64	49.49	65.27
Data-Guided Atlases 8192 FFDH Approx.	2.87	3.06	151.06	19.38
Data-Guided Atlases 8192 FFDH Approx.	**2.36**	**2.55**	35.71	88.98
Data-Guided Atlases 8192 FFDH Default	4.58	4.77	549.55	5.51
Data-Guided Atlases 8192 FFDH Default	2.37	2.56	37.23	85.37
Data-Guided Atlases 8192 FFDH Greedy	3.04	3.23	171.56	19.84
Data-Guided Atlases 8192 FFDH Greedy	2.39	2.58	37.26	85.67
Data-Guided Atlases 8192 MAXRECTS Approx.	3.52	3.71	296.53	9.98
Data-Guided Atlases 8192 MAXRECTS Approx.	2.41	2.60	40.41	78.66
Data-Guided Atlases 8192 MAXRECTS Default	5.02	5.22	644.32	4.66
Data-Guided Atlases 8192 MAXRECTS Default	2.45	2.65	43.22	81.07
Data-Guided Atlases 8192 MAXRECTS Greedy	4.08	4.28	420.15	7.59
Data-Guided Atlases 8192 MAXRECTS Greedy	2.44	2.63	43.78	75.99
Manual Atlases 4096 FFDH Approx.	3.43	3.63	259.81	11.48
Manual Atlases 4096 FFDH Approx.	2.45	2.65	48.43	61.95
Manual Atlases 4096 FFDH Default	3.44	3.64	266.18	11.18
Manual Atlases 4096 FFDH Default	2.52	2.71	52.11	57.77
Manual Atlases 4096 FFDH Greedy	3.46	3.66	262.99	11.47
Manual Atlases 4096 FFDH Greedy	2.48	2.67	49.69	60.78
Manual Atlases 4096 MAXRECTS Approx.	3.44	3.64	265.53	11.11
Manual Atlases 4096 MAXRECTS Approx.	2.55	2.74	53.44	57.46
Manual Atlases 4096 MAXRECTS Default	3.41	3.60	258.27	11.45
Manual Atlases 4096 MAXRECTS Default	2.54	2.73	52.46	57.75
Manual Atlases 4096 MAXRECTS Greedy	3.45	3.64	262.20	11.46
Manual Atlases 4096 MAXRECTS Greedy	2.49	2.68	51.58	58.81
Manual Atlases 8192 FFDH Approx.	3.44	3.64	267.32	11.05
Manual Atlases 8192 FFDH Approx.	2.52	2.72	51.05	58.95
Manual Atlases 8192 FFDH Default	3.49	3.69	264.29	11.35
Manual Atlases 8192 FFDH Default	2.52	2.71	52.30	57.28
Manual Atlases 8192 FFDH Greedy	3.46	3.66	262.15	11.53
Manual Atlases 8192 FFDH Greedy	2.50	2.69	49.33	61.34
Manual Atlases 8192 MAXRECTS Approx.	3.46	3.66	262.25	11.41
Manual Atlases 8192 MAXRECTS Approx.	2.50	2.70	51.04	59.62
Manual Atlases 8192 MAXRECTS Default	3.42	3.62	261.35	11.28
Manual Atlases 8192 MAXRECTS Default	2.55	2.75	53.24	57.28
Manual Atlases 8192 MAXRECTS Greedy	3.47	3.67	266.33	11.31
Manual Atlases 8192 MAXRECTS Greedy	2.56	2.76	53.58	57.58

Our team assembly running example in simulation 3 was unpredictable and this is reflected in the results in Table 5.7. Each transient atlases configuration performed better than its corresponding non-transient configuration. All transient atlas configurations had a CPU frame time of under three milliseconds and we observe that the transient data-guided atlases outperform the transient manual atlases, bu the prebuilt data-guided atlases do not outperform the prebuilt manual atlases. The manual atlases for simulation 3 consisted of the characters to be assembled, so these are better than the unpredictable team assemblies affecting the data of the prebuilt data-guided atlases. The separate textures with transient atlases and the FFDH packing algorithm had the least draw calls independent of the partitioning algorithm, because the separation allows for unrestricted combination and thus maximum preparedness for unpredictability.

Table 5.8: Average performance measurements of simulation 4 for each configuration. Highlighted rows indicate that transience was enabled.

Configuration	CPU Frame Time (ms)	GPU Frame Time (ms)	#Draw Calls	#Primitives per Draw Call
Separate Textures	12.11	12.79	1533.81	2.02
Separate Textures FFDH Approx.	4.98	12.46	5.18	1097.69
Separate Textures FFDH Default	4.94	12.53	4.50	1097.70
Separate Textures FFDH Greedy	5.02	12.50	4.91	1097.10
Separate Textures MAXRECTS Approx.	5.58	12.52	125.96	1013.69
Separate Textures MAXRECTS Default	5.71	12.84	124.87	1016.39
Separate Textures MAXRECTS Greedy	5.53	12.47	122.16	1017.32
Data-Guided Atlases 4096 FFDH Approx.	7.03	12.08	525.15	6.32
Data-Guided Atlases 4096 FFDH Approx.	4.69	11.85	11.47	467.05
Data-Guided Atlases 4096 FFDH Default	7.07	12.47	521.95	6.35
Data-Guided Atlases 4096 FFDH Default	4.54	11.91	7.30	470.84
Data-Guided Atlases 4096 FFDH Greedy	7.78	12.37	675.10	4.93
Data-Guided Atlases 4096 FFDH Greedy	4.72	11.85	11.99	467.78
Data-Guided Atlases 4096 MAXRECTS Approx.	7.45	12.20	602.57	5.51
Data-Guided Atlases 4096 MAXRECTS Approx.	4.63	11.78	14.38	369.91
Data-Guided Atlases 4096 MAXRECTS Default	7.83	12.32	698.91	4.76
Data-Guided Atlases 4096 MAXRECTS Default	4.57	**11.72**	7.16	470.97
Data-Guided Atlases 4096 MAXRECTS Greedy	8.95	12.28	941.70	3.55
Data-Guided Atlases 4096 MAXRECTS Greedy	4.66	12.00	16.63	370.40
Data-Guided Atlases 8192 FFDH Approx.	4.55	11.73	**3.01**	1099.10
Data-Guided Atlases 8192 FFDH Approx.	4.59	11.77	**3.01**	1099.07
Data-Guided Atlases 8192 FFDH Default	4.65	11.84	**3.01**	1099.09
Data-Guided Atlases 8192 FFDH Default	4.56	11.76	**3.01**	1099.11
Data-Guided Atlases 8192 FFDH Greedy	4.71	11.93	**3.01**	1099.15
Data-Guided Atlases 8192 FFDH Greedy	4.54	11.76	**3.01**	1099.08
Data-Guided Atlases 8192 MAXRECTS Approx.	**4.53**	11.73	**3.01**	1099.10
Data-Guided Atlases 8192 MAXRECTS Approx.	4.57	11.74	**3.01**	1099.16
Data-Guided Atlases 8192 MAXRECTS Default	4.66	11.85	**3.01**	1099.11
Data-Guided Atlases 8192 MAXRECTS Default	4.54	11.74	**3.01**	1099.06
Data-Guided Atlases 8192 MAXRECTS Greedy	4.75	11.94	**3.01**	1099.10
Data-Guided Atlases 8192 MAXRECTS Greedy	4.57	11.97	**3.01**	**1099.18**
Manual Atlases 4096 FFDH Approx.	9.53	12.62	1022.68	3.28
Manual Atlases 4096 FFDH Approx.	5.95	12.46	213.98	15.77
Manual Atlases 4096 FFDH Default	9.52	12.57	1028.25	3.26
Manual Atlases 4096 FFDH Default	5.89	12.56	207.50	15.91
Manual Atlases 4096 FFDH Greedy	9.47	12.53	1029.63	3.25
Manual Atlases 4096 FFDH Greedy	5.97	12.43	213.74	15.84
Manual Atlases 4096 MAXRECTS Approx.	9.56	12.58	1027.82	3.26
Manual Atlases 4096 MAXRECTS Approx.	6.03	12.66	215.59	15.71
Manual Atlases 4096 MAXRECTS Default	9.62	12.59	1032.78	3.24
Manual Atlases 4096 MAXRECTS Default	5.91	12.38	208.62	15.81
Manual Atlases 4096 MAXRECTS Greedy	9.61	12.63	1026.60	3.26
Manual Atlases 4096 MAXRECTS Greedy	5.99	12.42	216.14	15.66
Manual Atlases 8192 FFDH Approx.	9.66	12.74	1029.15	3.25
Manual Atlases 8192 FFDH Approx.	5.97	12.59	214.08	15.77
Manual Atlases 8192 FFDH Default	9.59	12.82	1025.03	3.27
Manual Atlases 8192 FFDH Default	5.89	12.59	206.61	15.98
Manual Atlases 8192 FFDH Greedy	9.69	12.95	1022.06	3.28
Manual Atlases 8192 FFDH Greedy	5.98	12.80	215.53	15.75
Manual Atlases 8192 MAXRECTS Approx.	9.62	12.64	1035.48	3.24
Manual Atlases 8192 MAXRECTS Approx.	5.98	12.42	215.57	15.72
Manual Atlases 8192 MAXRECTS Default	9.59	12.60	1030.88	3.25
Manual Atlases 8192 MAXRECTS Default	5.90	12.35	207.06	15.94
Manual Atlases 8192 MAXRECTS Greedy	9.74	12.79	1028.23	3.26
Manual Atlases 8192 MAXRECTS Greedy	5.94	12.42	214.01	15.73

In Table 5.8 we see the results for simulation 4 of dynamic textures. GPU frame times are higher for all configurations compared to other simulations and were only slightly reduced when the number of draw calls was decimated.

The configurations of manual atlases were significantly outperformed by the corresponding configurations of data-guided atlases except for the prebuilt data-guided texture atlases with a maximum texture atlas size of 4096 texels and the MAXRECTS and greedy packing and partitioning algorithms. Again the maximum texture size dictated performance for data-guided atlases without transience, although the packing and partitioning algorithms also had an impact on the number of draw calls.

5.3 Transient atlases overhead

We have chosen to zoom in on the first 25 frames of simulation 2 for checking the transient atlases overhead, specifically on separate textures and data-guided atlases with the FFDH packing algorithm and default partitioning. Simulation 2 is the most stable for its static usage of textures, so any overhead caused by atlas transience is best observed there. For data-guided atlases we picked 4096 as our maximum texture atlas size, because it leads to a higher number of smaller textures that can be combined when creating transient atlases, so the overhead is more visible. We compare the average performance measurements of all executions in figures 5.1, 5.2 and 5.3 for both prebuilt and transient configurations.

First we notice that transient atlases were used from around frame five and on in Figure 5.1 as the CPU usage dropped consistently for a few frames for transient configurations. In the peaks at frame four overhead in CPU time for the creation of transient atlases is observable if we compare the difference in CPU frame time for prebuilt and transient configurations for separate textures and for data-guided atlases.

In Figure 5.2 we see that the peaks in GPU frame time were similar and very high. The peaks were slightly higher for separate textures due to the overhead of having separate textures. We can see copying a large number of textures for the creation of transient texture atlases affected the GPU frame time significantly, because the GPU frame time transient separate textures configuration decreased slower after the peak than the transient data-guided atlases configuration. After frame five the GPU frame time kept a stable pace and was similar for transient configurations. For prebuilt separate textures the GPU frame time was a lot higher than for the other configurations.

Finally we confirm that transient atlases were indeed made around frame four and used from around frame five by checking the number of draw calls in Figure 5.3. The number of draw calls for prebuilt configurations was consistent with the use of separate textures or texture atlases. It was also lower for the the transient data-guided atlases at first compared to the separate textures with transient atlases, because of the use of separate textures and the time it took to create transient texture atlases.

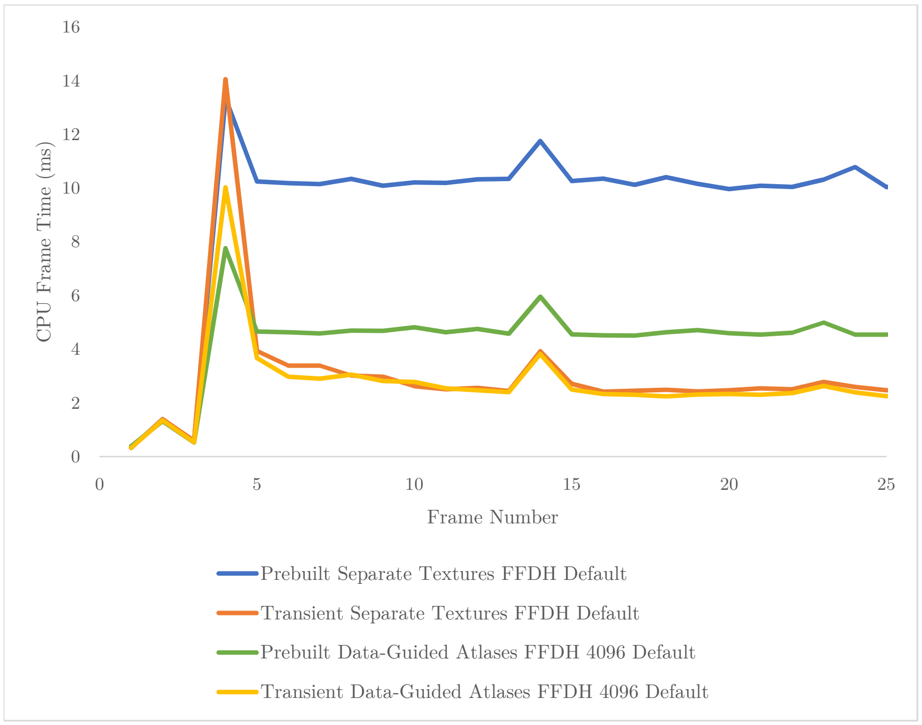


Figure 5.1: The average number of CPU frame time in milliseconds spent on the render thread over 25 frames for a selection of configurations ran with simulation 2.

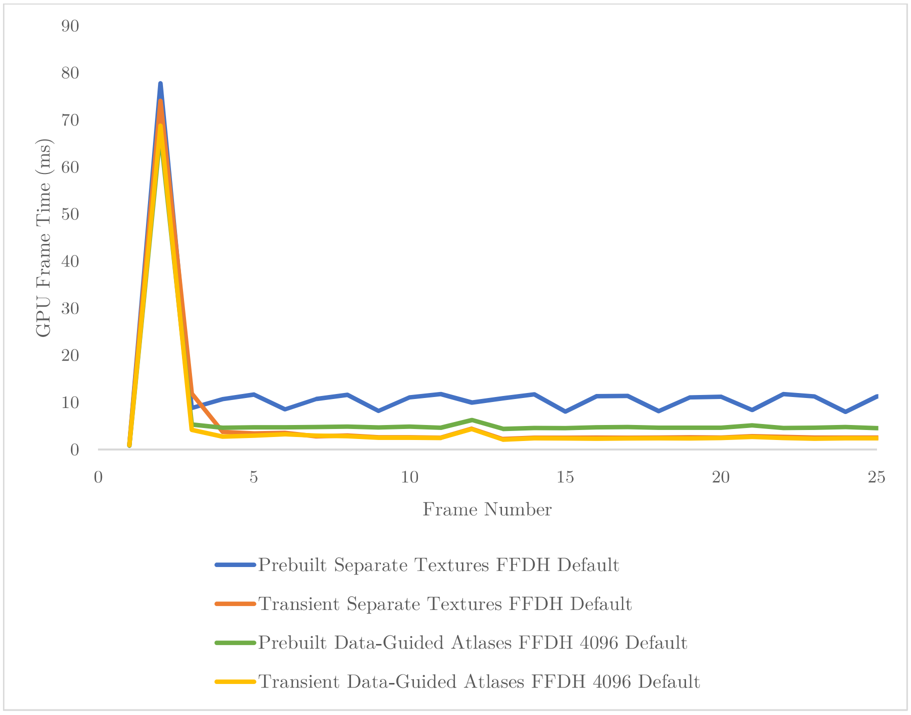


Figure 5.2: The average number of GPU frame time in milliseconds over 25 frames for a selection of configurations ran with simulation 2.

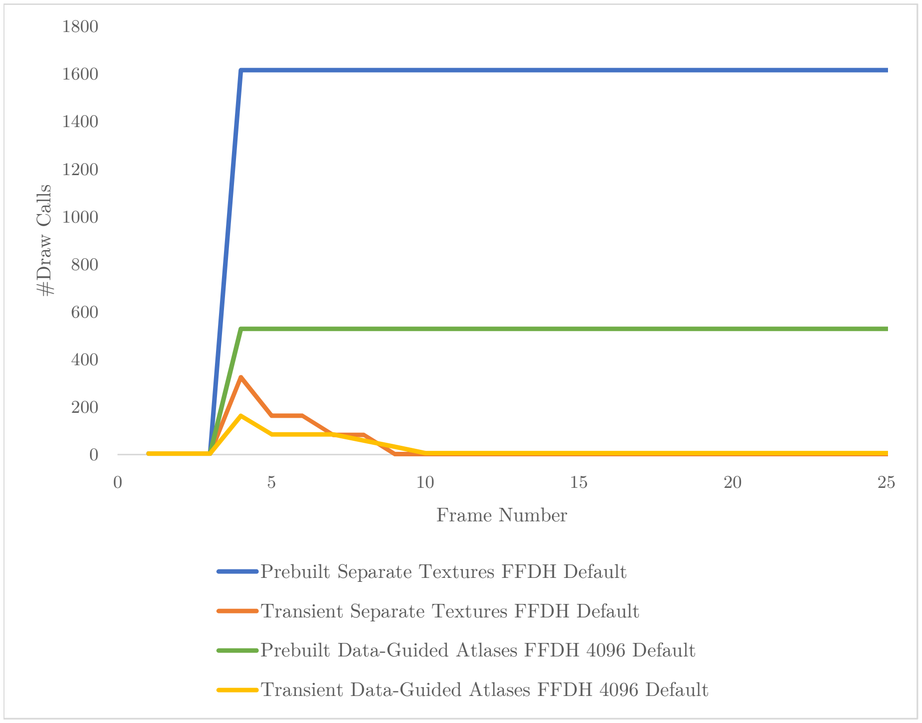


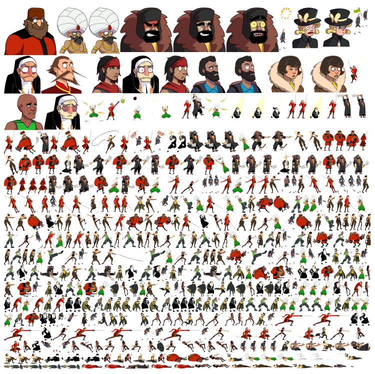
Figure 5.3: The average number of draw calls over 25 frames for a selection of configurations ran with simulation 2.

5.4 Memory usage

Table 5.9: Average area in square texels for the created texture atlases in simulation 3. Highlighted rows indicate that transience was enabled.

Configuration	Area (square texels)
Separate Textures	61175935
Data-Guided Atlases 4096	94876557
Data-Guided Atlases 4096	122935579
Data-Guided Atlases 8192	94014819
Data-Guided Atlases 8192	115571497
Manual Atlases 4096	105313384
Manual Atlases 4096	68763453
Manual Atlases 8192	105313384
Manual Atlases 8192	68562622

In Table 5.9 the area of the texture atlases that were created before or during simulation 3 is given. For separate textures with transience enabled the area is the lowest, because only the textures that are used in the run of the simulation are packed into transient atlases. For transient manual atlases the area is similar to the area of separate textures. However, for prebuilt data-guided and manual texture atlases the used area is much larger, although data-guided atlases use less memory than manual atlases. For prebuilt configurations, the area of all of the created atlases is measured and when using transient atlases only the textures in use are packed into transient texture atlases and then the area is much smaller, like for transient manual atlases. Unfortunately this is not the case for transient data-guided atlases, because texture atlases are loaded with lots of unnecessary textures. We chose five out of ten characters in simulation 3 and when those are spread out over multiple atlases, we get very high memory consumption for transient atlases. The spread of textures of multiple characters over multiple atlases for data-guided atlases can be seen in 5.4a compared to the manual single character atlas in 5.4b.


(a) Data-guided texture atlas.

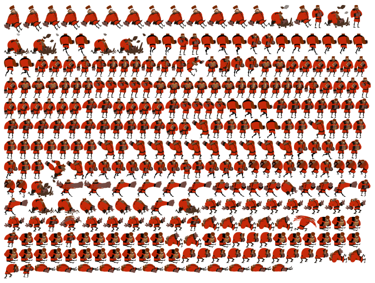

(b) Manual texture atlas.

Figure 5.4: Prebuilt texture atlases for simulation 3.

Chapter 6

Discussion

Our approach, implementation and experimental setup could have been done in many different ways. In this chapter we discuss and justify the approach and its implementation, evaluate the scale of the experiments, interpret the results, compare the configurations and speculate about the best use cases for the approach and its configurations.

6.1 Experiment scale

The experiments we conducted were not executed on large-scale simulations nor on a game, but on much smaller scale simulations. Our simulations used a huge number of textures, so that part was definitely large-scale, but the part where we switch scenes, levels or layers and travel through huge open worlds was simply not present.

6.2 Data-guided or manual composition

Most results have shown that we achieved reducing the number of draw calls using data-guided atlases and that data-guided atlases outperform manual atlases. Some results were not as good as for manual atlases, but at least came close, so we succeeded at automating the composition of texture atlases and matching performance of manual atlases for our experiments. However, the results of data-guided texture atlases are promising enough to believe that the approach is scalable with the right configuration and implementation and can benefit large-scale simulations and games. Prebuilt data-guided texture atlases seem especially suitable for textures of static geometry and textures that are used dynamically in a predictable manner. If textures are reused and combined a lot with different textures, the maximum atlas texture size should be lowered. Otherwise a larger texture atlas ends up being loaded for just one texture for example and batches might be broken unnecessarily. We did not take into account the reuse and combination of textures when processing the data gathered by the telemetry during rendering, so quite possibly there is much to be gained by grouping textures into atlas compositions based on those criteria, while also deciding the maximum texture atlas size automatically for the atlas composition to reduce draw calls on reuse and combination of textures and to reduce GPU memory usage.

6.3 Transient overhead

At first we thought the overhead of creating and using transient texture atlases would be much more severe, but the results were better than expected. Copying a large number of textures from one place in GPU memory to another is really fast, the copy texture region commands issued from the render thread only have a minor impact, also for separate textures and composing the textures into transient atlases with texture packing was performed on another thread to ensure smooth performance when that would take long. Both the process of texture packing and the process of copying textures to a transient texture atlas can be smeared out over multiple frames, which makes it suitable for real-time usage. Although the number of draw calls might be higher at first when it takes a little longer to transition to transient atlases, the performance is most likely always better, because textures always become part of larger texture atlases and creating larger texture atlases can only lead to an equal number of draw calls or less draw calls. The transition to transient texture atlases could also be masked by a loading screen and the number of draw calls is usually quite low if using prebuilt texture atlases instead of separate textures.

In our implementation we used offline rectangle bin packing for both prebuilt and transient texture atlases. The delay of building transient texture atlases could be reduced by applying online rectangle bin packing, especially when partitioning textures into different atlases. We did not choose to do so, because of the increased number of atlases necessary for online packed texture atlases[14]. We also did not use texture rotation in our packing algorithms for the sake of simplicity.

We have seen that the usage of transient texture atlases reduces draw calls when dealing with unpredictable combination of textures in our running example. Although separate textures with transient texture atlases often reduced draw calls as well as data-guided atlases, loading prebuilt separate textures takes much longer than loading prebuilt texture atlases and when a lot of large separate textures need to be combined, the transient atlas compositioning process takes much longer. When transient atlas usage is delayed for separate textures the initial frames are drawn with separate textures, which is really bad for performance as we have seen in our results.

6.4 Parameter configuration comparison

For a complete comparison we should have experimented with all possible configurations of course. We could have reduced the maximum texture atlas size even more to 2048 and combined different packing and partitioning algorithms for prebuilt and transient atlases. This was not feasible for this thesis, as it would have lead to a combinatorial explosion of configurations and results. Though we can compare the configurations that we did use in our experiments.

The partitioning of the textures into atlases also did not matter as much as we expected and this might be caused by the small scale and predictability of the simulations. If the simulation was of such a large scale that a lot of textures would benefit from being in one atlas, then the partitioning algorithm is expected to really make a difference. For data-guided atlases in simulation 3 the partitioning algorithm did matter and in some cases the greedy algorithm performs better and in others the approximation algorithm, either of them is better than the default. In simulation 4 the approximation algorithm performed better than the greedy algorithm.

6.5 Memory versus draw calls

Memory consumption for transient data-guided texture atlases was much higher than for transient manual atlases, because the manual atlases had less textures per atlas composition and were organized per character and because the textures for the chosen characters were spread out over the data-guided atlases. This means that the partitioning algorithms are insufficient for larger prebuilt data-guided textures or that the draw order interleaved textures of different characters, so that a good prebuilt data-guided atlas cannot be produced when different characters are combined. The partitioning could benefit from a smaller maximum texture size, such as 1024 or 2048, which enables more combinations for transient atlases and a higher chance that the characters are not spread out if the greedy or approximation partitioning algorithms perform well.

6.6 Requirements

Our approach to transient texture atlases requires the availability of fast in-GPU-memory compressed texture region mipmap copying to another compressed texture mipmap. We conducted our experiments only with the Direct3D 11 hardware API and the results might be different with for example OpenGL, Direct3D 12 or Vulkan.

Chapter 7

Related Work

Other approaches to improving draw call batching exist, such as clipmaps for terrain rendering, virtual textures and texture arrays for textures with the same properties. The alternative approaches are either not suitable for the general application of our approach or have certain drawbacks or advantages depending on the application and preferences or needs of the developer compared to our approach. In this chapter we discuss clipmaps, virtual textures and texture arrays and we review the contributions, usage, benefits and drawbacks of each approach.

7.1 Clipmapping

Tanner, Migdal, and Jones introduce the clipmap as a virtual mipmap[20]. A clipmap is a dynamic texture representation that partially clips the higher resolution levels of a mipmap and dynamically updates a clipped region of a mipmap level to greatly reduce memory usage. One of the goals was to support real-time rendering with a very large texture by only using a subset of the mipmap levels of that texture. This was accomplished by reducing the storage requirements for a very large texture by clipping the mipmap and the virtualization of texture memory. Dynamically updating a clipped region of a mipmap level requires that parts of the new region are present in memory, either on the same level or a lower resolution level. To accommodate for sudden scene changes, textures need to be prefetched from memory and cached. The necessary regions are put into a priority queue for loading, where coarser resolutions have a higher priority, because of the reduced bandwidth usage. This way a larger texture can be used and the batch is not broken by multiple smaller textures. Unfortunately a clipmap requires a spatial relation between texture data and geometry, because of the dynamically updated clipped region. The technique is therefore primarily used for terrain visualization. The clipmap approach was one of the firsts to introduce texture memory virtualization and texture prefetching, the foundation for virtual textures and texture streaming, an approach that has been developed further[7][18][20]. Opposed to clipmaps, Hüttner presents a grid of mipmaps to reduce memory usage by only loading higher resolution mipmap levels for specific textures in the grid, because a clipmap loads a whole mipmap level of a very large texture for only a specific region, which is rather costly memory-wise [12]. In the same year as [20] and [12], Igehy, Eldridge, and Proudfoot introduced prefetching in a texture cache architecture[13], reassuring the immediate need for a different solution for the storage and usage of textures.

7.2 Virtual textures

In [3] Barrett explains a CPU-like virtualization of texture memory. Barrett used the hints from John Carmack on the MegaTextures technique from id Software for guidance and dubs his technique sparse virtual textures. The literature prefers to refer to the technique as virtual textures, so we will also use this term, although AMD filed a patent in 2014 for partially resident textures [9]. The virtual textures technique utilizes the property of texture atlases that all the textures in one atlas can be drawn at once by putting all the mipmapped textures in one massive texture. To accommodate for such a large texture, the memory is virtualized and the texture only partially resides in GPU memory. When a texture needs to be drawn the texture coordinates are mapped onto the texture by a memory address translation. The requirement of real-time texture streaming to dynamically load and unload parts of the texture is a disadvantage. Sudden changes in a scene that require more detailed and different textures than currently in memory can lead to a blurry textured environment when the GPU bandwidth is limited, because it always loads the lowest mip first. To make the process of loading and visualizing textures smoother, priority queues and CPU-like cache policies must be used [4][5][10][16]. However, the GPU bandwidth and prioritization can still be insufficient and then low quality textures are drawn instead of high quality textures. Another disadvantage of such a massive texture is that the resulting file is very large. This makes the process of development when, for example, simply and quickly replacing a texture more complex and it also complicates applying a patch with texture changes to an already published application. Compared to our approach we only have some overhead on the CPU and might be limited by the GPUY bandwidth when loading textures and creating transient texture atlases, none of the other problems for virtual textures exist.

7.3 Texture arrays

Trapp and Döllner propose texture arrays[21], an alternative texture representation, which aggregates textures of the same format and resolution in arrays to reduce state changes. The approach also uses a data preprocessing step that analyzes the 3D geometry and texture usage beforehand. Although this technique shows promising results for a static city and could benefit from our approach, it requires textures of the same format and resolution and it is an enhancement rather than a solution for scenes that include static geometry, but also have more dynamic and unpredictable elements. Especially older hardware APIs and also some modern hardware APIs do not support texture arrays.

Chapter 8

Conclusion

We have shown that automating the composition of texture atlases is possible by gathering and processing texture rendering data into texture atlas compositions. Using data-guided texture atlases created from the automated atlas compositions we achieved similar or better performance than manually composed texture atlases by reducing the number of draw calls significantly. Not only have we shown that automation is possible, but also that it is a viable approach to reducing the number of draw calls, eliminating or at least decreasing the need for cumbersome manual composition of texture atlases.

When developing the approach for composing textures into transient texture atlases on the CPU and creating the transient texture atlases on the GPU at run-time we built upon the approach of automating the composition of texture atlases. Transient data-guided texture atlases enable the optimization of draw call batching by automatically combining textures, which themselves can be texture atlases, into texture atlases on load. The transient texture atlases can utilize the full maximum texture dimensions of the GPU and are created based on data and suitable for dynamic, unpredictable and emergent texture usage. The overhead turns out to be far less than the performance gain and allows real-time rendering.

We recommend transient data-guided atlases with a maximum texture atlas size depending on the usage of textures. The maximum texture atlas size should be lower when textures are not used statically, used unpredictably or reused a lot. Unfortunately, we cannot recommend our partitioning algorithms, because of high memory consumption. We do believe there are better algorithms out there. Nevertheless we only recommend composing texture atlases manually when the way textures are drawn is straightforward and predictable and when creating all the atlas compositions manually is not cumbersome.

Finally we predict large-scale simulations with many 2D textures such as the simulation of a virtual 3D city in [21] can benefit from using transient data-guided texture atlases.

8.1 Future work

Our implementation was not flawless, our approach can be improved and both can be extended in the future.

Near the end of our research we gained new insights on texture coordinates and size alignment for mipmaps and compression that we incorporated into the work, but that also paved the way for adaptive transient data-guided texture atlases. These texture atlases would be able to adapt based on rendering data gathered at run-time and immediately processing the data to adapt the texture atlases by moving texture regions in an atlas to another texture atlas instead of only combining textures or texture atlases into larger transient texture atlases. After some testing in AbbeyCore we thought this was not possible or not viable due to the memory wastage of aligning textures for the compression block size and the number of mipmap levels, although we did want to pursue adaptive transient data-guided atlases at first. Adding adaptation to the transient data-guided texture atlases could reduce draw calls even more, because of maximum flexibility in composing the textures and could eliminate the need for gathering rendering data beforehand.

Currently textures are only put into a single texture atlas, there are no duplicates. For textures with a high rate of usage and combination with lots of different textures putting them into multiple texture atlases, and introducing duplicates, could reduce the number of draw calls. Textures would need to be classified based on criteria that indicate duplicates would be beneficial and when rendering such textures, a texture change has to be avoided by checking if the currently bound texture (atlas) already is or contains the texture to be rendered to reduce draw calls instead of changing the texture to a texture atlas that also contains a duplicate of the texture and introducing a new draw call.

Grouping textures based on different kinds of usage might in general be a good idea. Textured static geometry is always rendered the same way, so draw call batching can be optimized easily before run-time. Dynamic and unpredictable usage is hard to optimize before run-time and those textures could benefit from being put into other texture atlases than those for static textures. Classification based on usage could be done by applying machine learning techniques on the rendering data. Optimizing a classification model for textures would be interesting and may reduce draw calls.

The high memory consumption for combining data-guided texture atlases into transient texture atlases for unpredictable use needs to be researched further. If our graph partitioning algorithms were not suitable, clustering for community detection in graphs[8] could improve partitioning many textures into multiple atlases to improve draw call batching and lower the high memory consumption.

Our experiments used many textures, but were not necessarily large-scale in the sense of dynamicity and unpredictability. This would have made a stronger case for our approach, so the results of applying transient data-guided texture atlases to a large-scale game or simulation would be interesting.

Bibliography

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining Association Rules Between Sets of Items in Large Databases". In: *SIGMOD Rec.* 22.2 (June 1993), pp. 207–216. ISSN: 0163-5808. DOI: 10.1145/170036.170072.

[2] Konstantin Andreev and Harald Räcke. "Balanced Graph Partitioning". In: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '04. Barcelona, Spain: ACM, 2004, pp. 120–124. ISBN: 1-58113-840-7. DOI: 10.1145/1007912.1007931.

[3] Sean Barrett. "Sparse virtual textures". Game Developers Conference (GDC). 2008. URL: https://silverspaceship.com/src/svt/.

[4] Matthaus G Chajdas et al. "Virtual texture mapping 101". In: *GPU Pro: Advanced Rendering Techniques*. AK Peters/CRC Press, 2010, pp. 185–195.

[5] David Cline and Parris K Egbert. "Interactive display of very large textures". In: *Visualization'98. Proceedings*. IEEE. 1998, pp. 343–350.

[6] Patrick Cozzi and Christophe Riccio. "Improving Performance by Reducing Calls to the Driver". In: *OpenGL Insights*. AK Peters/CRC Press, 2012.

[7] Anton Ephanov, Chris Coleman, and TX Richardson. "Virtual texture: A large area raster resource for the GPU". In: *The Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*. 2006.

[8] Santo Fortunato. "Community detection in graphs". In: *Physics reports* 486.3 (2010), pp. 75–174.

[9] Tom Frisinger. *Partially resident textures*. US Patent 8,907,969. Dec. 2014.

[10] Charles Hollemeersch et al. "Accelerating virtual texturing using CUDA". In: *GPU Pro: advanced rendering techniques*. Vol. 1. CRC Press, 2010, pp. 623–641.

[11] John Hopcroft and Robert Tarjan. "Algorithm 447: Efficient Algorithms for Graph Manipulation". In: *Commun. ACM* 16.6 (June 1973), pp. 372–378. ISSN: 0001-0782. DOI: 10.1145/362248.362272.

[12] Tobias Hüttner. “High resolution textures”. In: *Visualization’98-Late Breaking Hot Topics Papers* (1998), pp. 13–17.

[13] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. “Prefetching in a texture cache architecture”. In: *Proceedings of the ACM SIGGRAPH / EUROGRAPHICS workshop on Graphics hardware*. ACM. 1998, 133–ff.

[14] Jukka Jylänki. “A thousand ways to pack the bin - a practical approach to two-dimensional rectangle bin packing”. 2010. URL: http://clb.demon.fi/files/RectangleBinPack.pdf.

[15] Manny Ko. “A Fast and High-Quality Texture Atlasing Algorithm”. In: *Game Engine Gems 3*. CRC Press, 2016.

[16] Martin Mittring et al. “Advanced virtual texture topics”. In: *ACM SIGGRAPH 2008 Games*. ACM. 2008, pp. 23–51.

[17] *Renowned Explorers: International Society*. [Windows, Mac OS, Linux]. Utrecht, The Netherlands: Abbey Games, 2015. URL: renownedexplorers.com.

[18] Antonio Seoane et al. “Hardware-independent clipmapping”. In: *WSCG2007, International Conference in Central Europe on Computer Graphics, Czech Republic*. Václav Skala-UNION Agency, 2007.

[19] Mechthild Stoer and Frank Wagner. “A simple min-cut algorithm”. In: *Journal of the ACM (JACM)* 44.4 (1997), pp. 585–591.

[20] Christopher C Tanner, Christopher J Migdal, and Michael T Jones. “The clipmap: a virtual mipmap”. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM. 1998, pp. 151–158.

[21] Matthias Trapp and Jürgen Döllner. “Geometry Batching using Texture-arrays.” In: *GRAPP, the International Conference on Computer Graphics Theory and Applications*. 2015, pp. 239–246.

[22] JMP Van Waveren. “Real-time DXT compression”. Tech. rep., id Software, Inc. 2006.

[23] JMP Van Waveren. “Real-time texture streaming & decompression”. Tech. rep., id Software, Inc. 2006.

[24] Lance Williams. “Pyramidal parametrics”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 17. 3. ACM. 1983, pp. 1–11.

[25] Matthias Wloka. “Batch, Batch, Batch: What Does It Really Mean”. Game Developers Conference (GDC). 2003. URL: https://www.nvidia.com/docs/IO/8228/BatchBatchBatch.pdf.

[26] Matthias Wloka. “Improved Batching via Texture Atlases”. In: *ShaderX3: Advanced Rendering with DirectX and OpenGL*. Vol. 44. Charles River Media Inc., Hingham, Massachusetts, USA, 2004.

[27] Alan L Yuille and Norberto M Grzywacz. “A computational theory for the perception of coherent visual motion”. In: *Nature* 333.6168 (1988), pp. 71–74.