WHITEPAPER

GRANITE SDK 5.0

TEXTURE STREAMING MIDDLEWARE

GRAPHINE

+32 9 247 01 05

graphinesoftware.com

info@graphinesoftware.com

GRANITE SDK

TRADEMARKS

COPYRIGHT

2018

CONTENTS

Inroduction

This document introduces you to some of the features and techniques of advanced texture streaming and our product Granite SDK, version 5.

Granite SDK is a middleware solution that handles fine-grained texture streaming for real-time 3D software applications. It consists of both production and runtime components. The runtime component can be integrated into any 3D or game engine. While running your application, it will analyze which texture data is visible to the virtual camera in your scene, and continuously load the required texture data asynchronously from disk into video memory.

With Granite SDK, there's no need to load all textures up front at the start of the application. All the runtime subsystems are highly optimized to ensure the lowest latency possible between requesting a piece of texture data, and the moment it becomes available in video memory for rendering. It doesn't matter how many textures you have in your scene, or how high the resolutions of those textures are, Granite SDK will elegantly handle all your content at 90 frames per second easily.

The Granite runtime runs on millions of devices worldwide on multiple platforms. Its production pipeline is industry proven by some of the leading game production studios such as Wargaming, Sumo Digital, and Funcom. Granite SDK is developed by a dedicated team, is available with premium support, and custom features can be added on request.

The following benefits are provided by Granite SDK:

- Use 4K or 8K textures in all your materials
- Support 4K rendering
- Lower your texture memory requirement
- Decouple memory usage from the assets and their resolution in your scene
- Reduce startup loading times to seconds
- Free artists from technical constrains
- Get fine-grained control over the runtime resources
- Have textures of up to 256.000 by 256.000 pixels in size
- Have single hero assets beyond 16K by 16K or hundreds of UDIM patches
- Have dynamic and user authored environments combined with thousands of unique assets
- Optimize memory for ultra-high resolution procedural textures (2D textures and cubemaps)
- Automatically get the optimal texture quality for a fixed amount of memory that you define
- Select the optimal performance versus memory trade-off for each texture
- Enable ultra-high texture quality on medium to low spec devices
- Increase artist iteration time with our fast texture encoder
- Support every texture format and compression in your engine
- Minimizing texture state changes in your renderer to improve performance
- Compress your textures by 60% and more

- Optimize and finely tune disk IO
- Support ultra-dense, compact scenes and large open worlds with one system

The remainder of this document first discusses some of the key abilities of the Granite SDK, next, what comes in the box, and finally, has some words on integrating Granite SDK into your own application and extra features.

Streaming with Granite SDK

Texture streaming is the process of progressively loading a texture to memory. By progressively loading textures, loading times can be shortened (as not all textures have to be loaded up front) and memory usage can be reduced (as not all texture data needs to be present in memory at once). In turn, memory reduction allows the application to use and display more textures, increasing the visual quality. The biggest advantage is that the total GPU memory can be used a lot more efficiently than with traditional textures.

For example, a streaming system would allow you to render multiple characters on screen each using its own unique 8192x8192 texture. Characters in the virtual distance would not have their full texture loaded into memory, maybe only the top mips. Once the virtual camera moves closer, the full resolution textures would be loaded and displayed on the fly. A more advanced streaming system would even differentiate between the orientations of the virtual camera. If the backside of a character is not visible, its textures would not be loaded until the camera moves around the character.

Granite SDK is an advanced streaming middleware library. It allows you to load textures asynchronously during the course of the application without needing to load all the textures up front. To accomplish this, the Granite SDK streaming system manages texture data in small tiles (typically 128x128 texels). Tiles are the basic units of streaming in the Granite SDK, each tile can be requested and loaded separately by our library, resulting in a texture that is sparsely loaded. Multiple resolution levels (mip maps) of your textures are tiled as well, so for example, one tile can hold detailed pixels from a mountain top, and another tile can hold a low-detailed version of the entire landscape. Figure 1 shows this.

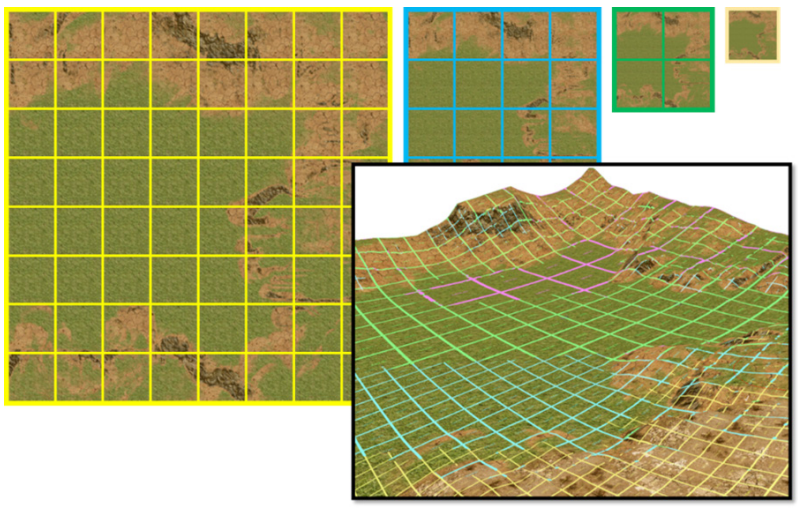

Figure 1: Textures are mip mapped and split into tiles, the basic units for streaming.

Classic Texture Streaming

Granite supports two types of streaming modes. First, Granite SDK allows for classic mip map streaming: memory for a texture's mip map pyramid is fully allocated on the GPU, but individual mips are progressively streamed as they are used. Figure 2 shows this. Textures reside on the hard disk, and are split into tiles. Texture mips are streamed by loading tiles from the disk and uploading them to the GPU. While this approach doesn't save much GPU memory in itself, the cost of loading textures is significantly reduced, allowing you to swap textures in and out more easily. This approach is ideal if you simply want to reduce loading times and bandwidth use for your textures.

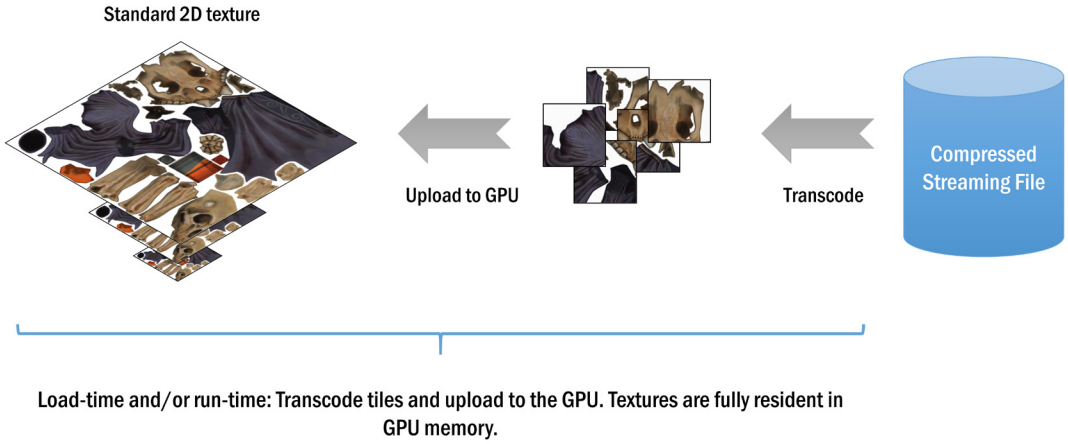

Figure 2: Using the Granite SDK run-time for classic texture streaming.

Fine-Grained Streaming by Virtual Texturing

Granite SDK's second streaming mode gives you more fine-grained control over what texture data is resident in video memory by managing individual small tiles of texture data. This allows you to use an extremely high amount of texture data (each texture atlas can be up to 256K by 256K) that are orders of magnitude larger than what can be stored in (video) memory. This approach is not only ideal for large landscapes, every scenario with high-quality textured meshes benefits from this approach.

The core of this system replaces the traditional fully allocated GPU texture (see the previous Section 'Classic Texture Streaming' or more info on our website) with a cache texture that holds tiles (see Figure 3). Reading from such a partially resident texture is performed by a shader that translates your input texture coordinates into texture lookups in the cache texture. Even when using moderately-sized textures, the fine-grained control provided by the Granite SDK results in less video memory required for your texture data compared to a typical mip map based texture streaming solution. This approach is typically called Virtual Texturing, it can be seen as a successor to the MegaTexture technology of id Software. More information on Virtual Texturing can be found on our website.

To determine what tiles to load into the cache texture, the system first determines what tiles are visible on the screen and at what mip map resolution they should be loaded. This works fully automatically and does not require manual tweaking or tagging of objects. Any tiles not present in the cache are then streamed from the storage device, transcoded, and uploaded to the GPU cache texture.

A big advantage of this approach is that it is inherently scalable, as texture tiles are only loaded if their content is visible on the screen and are loaded at the required mip map resolution. It automatically loads less data on smaller screen resolutions requiring less manual tweaking or user configuration. Basically, it imposes no limits on the texture resolution anymore, your application can have as many textures as necessary to reach your quality goals.

Streaming from Tile Sets

Granite SDK streams any texture type, whether it consists of typical PBR-style materials, photogrammetry-generated textures, UDIM textures, cube maps, environment maps, displacement maps, and many more. Granite SDK typically streams textures from one or more Tile Sets. A Tile Set bundles several textures in a big atlas that can contain many, possibly thousands of textures. Figure 4 illustrates this. A Tile Set can have multiple layers. Layers allow bundling together different textures belonging to the same mesh texture coordinates in a single logical unit. For example, assume you have a character with three painted texture layers: diffuse, normal, roughness. Then Granite allows you to bundle these three textures together in a single, what we call, Stacked Texture with three layers. Stacking textures this way offers significant benefits in streaming performance since Granite knows how the three source textures are related to each other (they share texture coordinates). Granite then ensures that the data of those three layers can be efficiently accessed as a single unit.

All textures in the Tile Set are split into tiles which are served to the streaming system. A Tile Set can be stored on disk or can be procedurally generated in memory on-the-fly. Both Tile Set files and procedural Tile Sets feed tiles to the Granite Streaming Runtime. The Granite Runtime in turn uploads these tiles to the VRAM, and the same Runtime does this for both the classical mip map streaming and Virtual Texture streaming scenarios. Figure 5 shows this.

If a Tile Set is stored on disk, it consists of multiple files (GTS header and GTP page file(s), more on this later in this text). The lay-out of each of the Tile Set's files is carefully designed for high data throughput from disk into memory. Tile Set files can be optimized for different storage devices taking the device's specific characteristics (e.g. cluster size) into account.

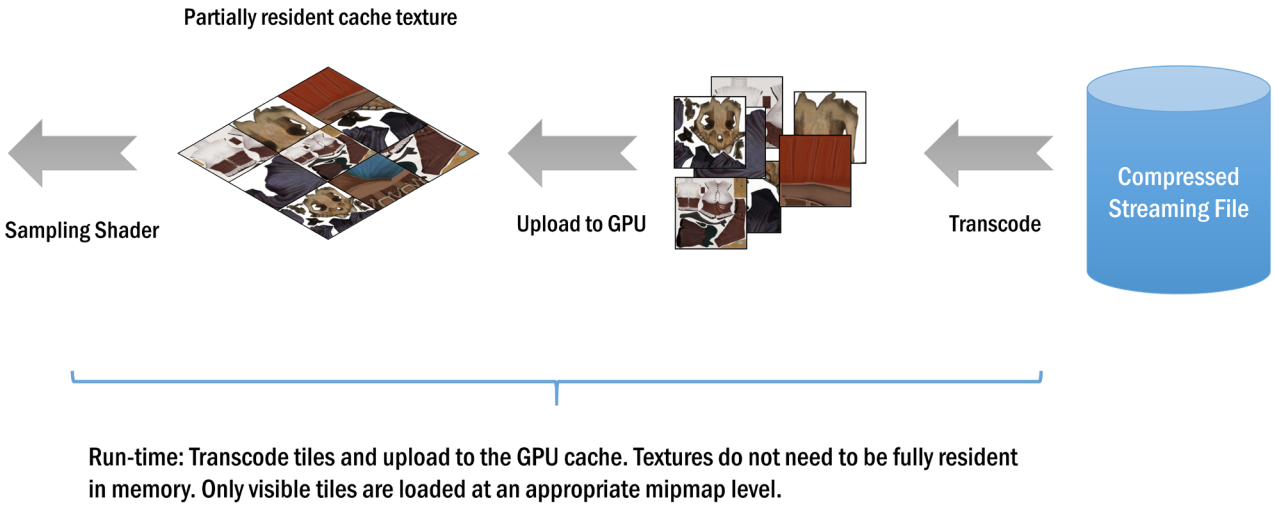

Figure 3: Using the Granite SDK run-time for fine-grained streaming (a.k.a. Virtual Texturing).

During streaming, tiles travel through multiple levels of tile caches, from hard disk to system memory, to video memory. Each cache can be individually configured in size. Such a system ensures the low latency access to the texture data while giving complete control on the amount of resources used at runtime.

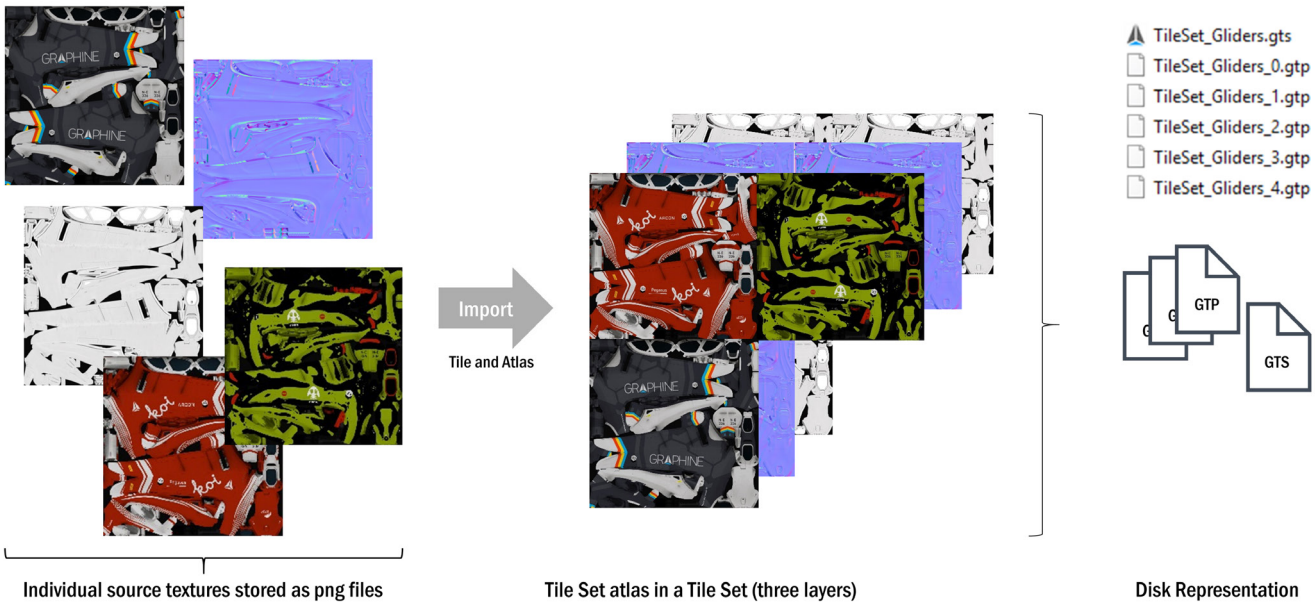

Figure 4: Tile Set file combines many textures in many layers.

Streaming Procedurally-Generated Tiles

Granite SDK supports procedural generation of tiles, sometimes called Procedural Virtual Texturing. Instead of loading a tile from a Tile Set file on hard disk, the tile is created on-the-fly and supplied of its pixel data by a user-implemented call back. This generation of pixel data is done at runtime, on the end user's system, pixels are never stored on disk.

The application provides either raw data (e.g., RGBA) or compressed data (e.g., BC7). A generated tile is placed in the caches just as offline-generated file-loaded tiles are. Because of the typical coherency between rendered frames, procedurally-generated tiles most often stay in the cache for quite a while. It isn't until they get evicted and later requested again, that they get asked to be regenerated by the procedural system. Procedural VT can be used for custom terrain splatting, dynamic tessellation or height maps, decals, procedural level generation, and many more scenarios.

Optimizing Memory Consumption

Streaming textures instead of loading them in advance reduces memory consumption and makes memory consumption more predictable, even when working with very large textures and large amounts of textures. A streaming system loads only those textures that are necessary, so your texture memory budget is not determined anymore by the sum of all texture sizes, but by the amount of textures visible on screen. In other words, you can use more textures than your physical memory would allow. Granite's Virtual Texturing mode goes one step further and loads only those texture tiles that are visible on screen. This means Granite loads textures into memory in a much more precise and fine-grained fashion, so less memory is wasted.

Granite SDK gives you control over the memory consumption by allowing you to control two caches, a CPU and a GPU cache. The CPU cache sits between the Tile Set file on the hard disk and the GPU: tiles are read from disk and put in this cache before they are uploaded. The GPU cache sits in the GPU VRAM and consists of GPU textures where tiles are uploaded to. You decide how many caches are set up for each datatype for a Tile Set and provide a texture budget in number of megabytes.

Teams typically set up the cache sizes to correspond with their graphics quality preset of their application. Low end systems run with a small cache, high-end systems with a large. Performance scales with the cache sizes, but even low cache settings can give great quality for some applications. Some teams, for example, Funcom with their game Conan Exiles , even set their caches as low as 128MB for CPU and 128MB for GPU. This way, they can still render high resolution textures on low-end systems without any visual artefacts. For high-end systems, Funcom for this game set their caches to 512MB CPU and 512MB GPU.

In case the cache is set too small, cache thrashing can occur. Tiles keep getting loaded into the cache, evoked, only to be reloaded again afterwards. Granite SDK supports a mechanism to prevent cache thrashing. Automatic LOD biasing lowers the load on the streaming system by suspending requests for higher resolution textures as long as cache thrashing occurs. It allows the system to recover by gradually requesting more detailed textures over time. Such a system, for example, can adjust to a sudden spike in hard disk activity and allow the hard disk to recover.

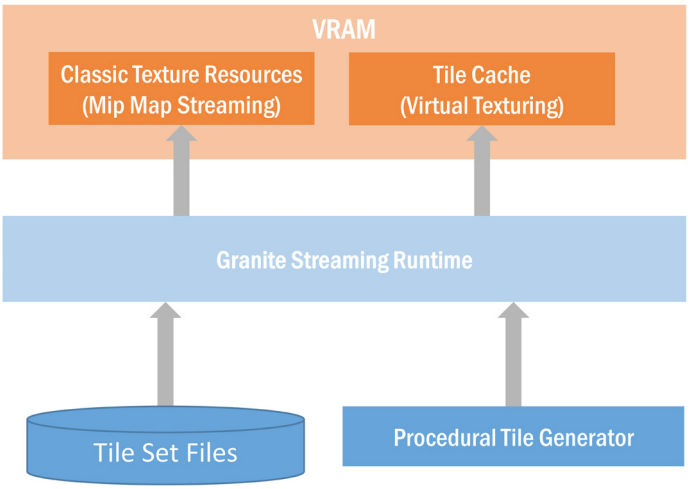

Figure 5: Tiles are loaded from disk or procedurally generated and streamed to VRAM for both mip map streaming and Virtual Texturing.

The Granite SDK Library Overview

Out of the box, Granite SDK comes with a streaming runtime library, tile creation and management tools, and extensive online documentation and samples.

Runtime and Platforms

The runtime library exposes a cross-platform C++ API. It runs highly-optimized, multi-threaded SSE code to ensure the fastest low-latency access to tiles possible. It is available as both 32-bit and 64-bit libraries and comes with a number of so called tiling back ends, platform-specific sub libraries targeting the various available 3D graphics APIs. Granite SDK includes tiling back ends for Direct3D11, Direct3D12, PS4, OpenGL, and OpenGL ES (Direct3D9 only upon request). These back ends are highly optimized to ensure optimal GPU performance. Each back end is customized to take advantage of the available features a specific 3D API exposes. For example, some run compute shaders, others run pixel shaders or fall back to run on the CPU.

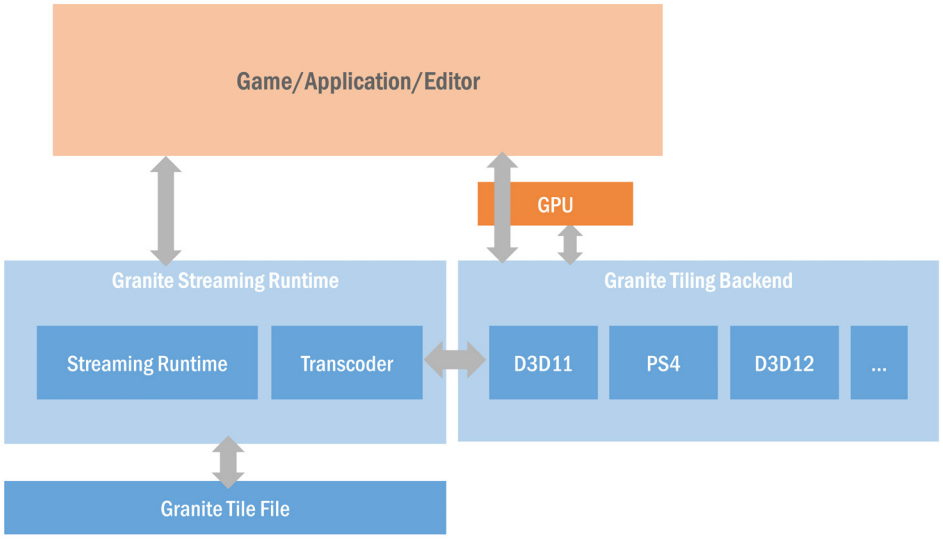

Figure 6: Overview of the Granite SDK Runtime

Granite SDK runtime supports many platforms: Windows (x86, x64), MacOS, Linux, PS4, PS4 Pro, XBOX One, XBOX One X, and Android. Each platform-specific runtime is customized to take advantage of platform-specific features including threading and IO APIs, custom texture format upload routines, and platform-specific GPU APIs. Note that Granite SDK does not support Partially Resident (PRT) hardware textures or Tiled Resources (TR) anymore. Support[1] for PRT and TR was ceased as later versions of Granite's software VT system surpassed all benefits of PRT or TR. Also note that the Granite SDK Tile Set building tools run on Windows x64 only.
The newest VR and AR platforms are also supported by the runtime. As many of the teams using Granite have proven, Granite SDK has no issues with driving texture streaming for virtual reality use cases and is fully compatible with VR SDKs such as Oculus SDK and SteamVR SDK. Clients using Granite for VR include Survios, Sólfar, Nurulize, and Realities IO.

Transcoding and Compression

Granite SDK comes with a built-in transcoder: a coding unit that decompresses tiles from our highly compressed proprietary disk format straight to a compressed GPU texture. Modern graphics hardware typically stores textures compressed in video memory to save bandwidth and maximize the amount of texture data that can be used. Streaming benefits from these formats as well, they save bandwidth when copying textures from system to GPU memory. The transcoder is highly optimized for a low CPU impact and uses SIMD optimized code. Granite SDK supports almost all frequently used compressed texture formats: BC1, BC3, BC4, BC5, BC6, BC7, and ASTC.

Although GPU compression formats are perfect for GPU sampling, they are not optimal with respect to the size of textures on disk. Granite SDK additionally compresses Tile Sets on disk with a proprietary texture codec specifically designed for common texture content used in (real-time) rendering engines, including but not limited to color data, alpha channels and normal vectors.

These codecs achieve a much higher compression ratio (at comparable texture quality) than commonly used fixed-rate compression techniques (e.g. DXT1, DXT5 ...). At runtime, tiles compressed with these codecs get transcoded on-the-fly to a GPU-friendly fixed-rate format. Additionally, the Graphine codecs allow you to adjust a quality setting so that you can control the storage size and quality for a specific texture. Some scenarios show a reduction of disk space used by textures by up to 86% compared to DXT5. You can also opt for a high quality scheme that codes textures directly to specific BC format on disk (e.g., BC5 on disk). This does not require transcoding of the texture to a GPU format, which saves processing time and allows for the best offline BC compression quality. Finally, for those scenarios requiring the highest quality, Granite also supports lossless coding.

---

[1] For more information on PRT and TR, see Graphine's presentation together with Microsoft at Build 2013 on Tiled Resources here: https://www.youtube.com/watch?v=QB0VKmk5bmI . Additional information can be found on our website http://graphinesoftware.com/our-technology/virtual-texturing.

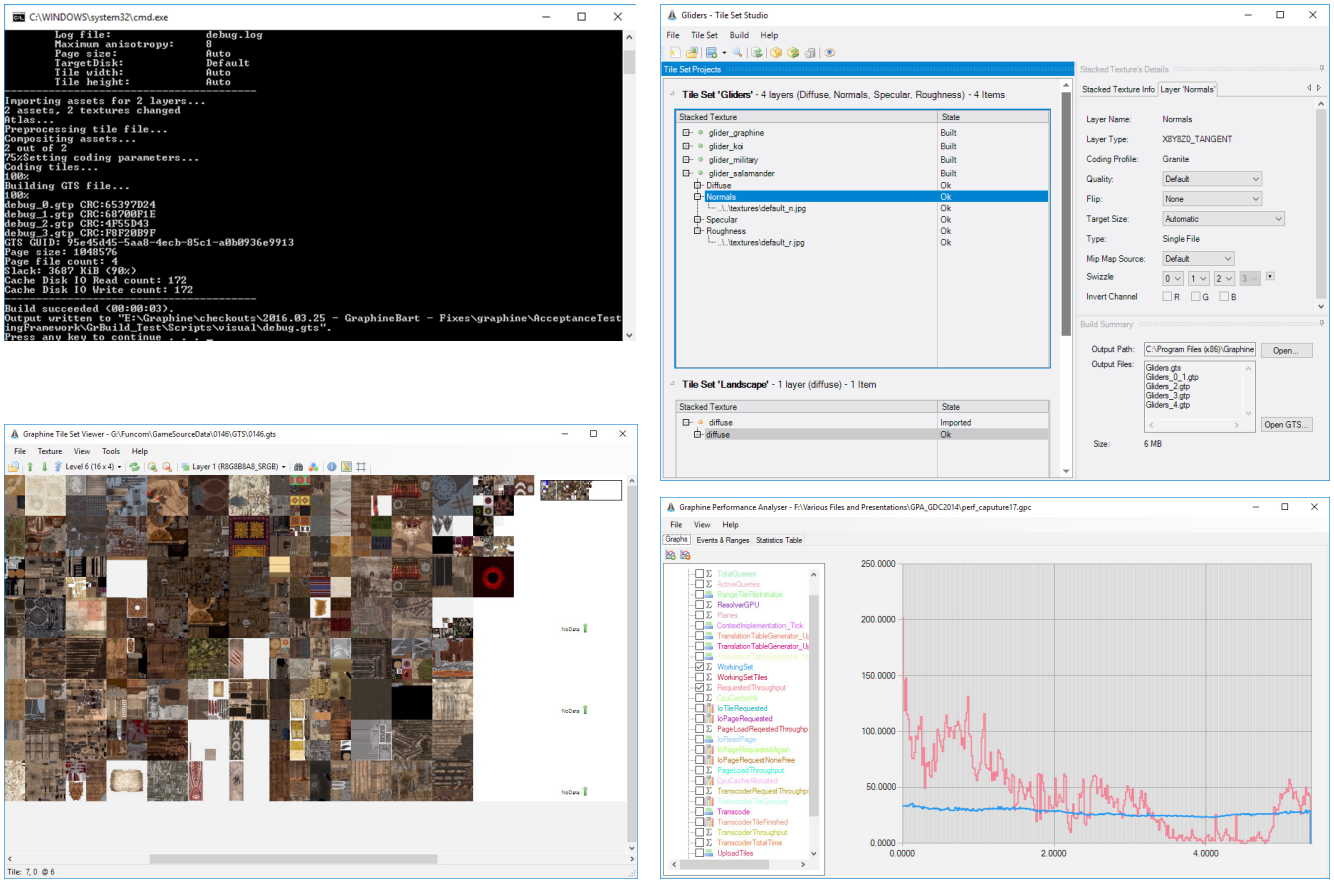

Figure 7: Screenshots of the Granite SDK tools: upper row, GrBuild CLI and Tile Set Studio GUI build tools, respectively, lower row, Tile Set Viewer and PerformanceAnalyzer inspection tools.

Creation Tools

Granite SDK comes with tools to create, manage, and inspect Tile Sets, as well as to profile streaming performance (see Figure 7). Creating and managing is performed by a command-line interface tool, GrBuild, or a GUI-capable tool called Tile Set Studio. Both tools offer the same functionality: they create a Tile Set by importing texture data, splitting them into tiles, coding the pixel data, and writing the streamable Tile Set file to disk.

All common image file formats are supported (e.g., .bmp, .jpg, .jpeg, .gif, .pcx, .png, .tga, .psd, .tif, .tiff, .hdr, .dds, .exr), and during import, the tools can perform basic image operations such as resizing and mirroring. Individual channels of images can be inverted, swizzled, and mixed and matched from different input textures, e.g., filling a X8Y8 layer with two individual grayscale images. Importing tiled images is supported (e.g., multiple images describing columns and rows of a large image) including the well-known UDIM texture format which the streaming runtime natively supports. Linear and SRGB pixel data are supported, as well as HDR, and, if preferred, the tools can perform bit depth conversion from HDR to LDR and vice versa. Cube maps can also be imported and are supported by the run time. After importing, textures are coded, at which stage, if preferred, each texture can have its compression settings individually set.

After coding, the tools atlas each individual texture in a large texture atlas. Neither the tools nor the Granite SDK Runtime impose a real limit on the atlas resolution. The Tile Set's texture atlas can grow without issues to 256K x 256K texels.

The Granite SDK also comes with a viewer tool for Tile Sets, Tile Set Viewer, which is typically used as an inspection tool at production time for GTS files. Finally, Granite SDK comes with a performance analyzer tool that allow inspection of key streaming performance metrics of the streaming application.

Integration and Workflow

Engine Integration

The Granite SDK is designed from the ground up to be easy to integrate into existing engines and pipelines. It supports common practices such as easy to overload memory allocations and error reporting. Besides this, it also allows fully customizing its use of threads by integrating with your existing job system. Granite has a standard multithreading implementation if you don't have a job system in your engine. Besides threading, (asynchronous) disk-IO can also easily be customized, allowing the Granite SDK to be used alongside your in-house streaming system.

When integrating Granite SDK in your engine we understand that you want maximal control about Granite's communication with the graphics API with a minimal amount of integration effort. Therefore, Granite SDK offers three levels of integration between Granite and the Graphics API that vary in their degree of abstraction and code that needs to be written by the user:

- Fully automatic: You provide Granite SDK with a pointer or handle to your graphics device context and let the library handle everything for you. Granite SDK will provide you with standard graphics API texture objects which can be used in your shader the same way as any other graphics API texture object.
- Engine managed resources: Granite will call back to the engine to create/destroy resources but will itself ensure that the resource contents are correctly updated and maintained.
- Engine API: The game engine provides an implementation of all the graphics API callbacks used by Granite. This provides maximal flexibility on how the Graphics API is accessed at the expense of more integration work.

Fetching texels from Granite's Stacked Textures is done in your shaders through specialized Granite shader instructions. These are available for HLSL (old and new syntax) and GLSL, and for shader models SM3, SM4, and SM5. All type of shaders are supported: pixel, vertex, geometry, compute, and tessellation shaders. Integrating into your custom shader involves replacing the typical tex2D with a line like this:

```
float4 output;
Granite_Sample_HQ(grConstantBuffer, grLookupData, cacheTex, 0, output);
```

The Granite shader instructions are provided through Granite headers which you can include in your shaders.

In a Virtual Texture streaming scenario, Granite SDK must determine what tiles to load into the cache texture by looking at what tiles are visible on the screen and at what mipmap resolution they are needed. Granite SDK exposes multiple methods to accomplish this, all working fully automatically, not requiring any manual tweaking or tagging of objects. The recommended method binds an extra MRT render target during regular rendering which the Granite SDK shader code fills. Another method binds a Read/Write texture that is a fraction of the screen size. The texture is again written during regular rendering by Granite SDK shader code. The final method, the simplest and most portable option, re-renders a scene at a lower resolution using a special Granite resolve shader.

Configuring for optimal performance and memory consumption

Granite exposes a lot of options that allow you to tweak Granite for different target hardware and usage scenarios. For example, Granite gives full control over its caches. You can assign caches to Tile Sets, share caches between Tile Sets and data types (e.g., one BC7 cache). You set the texture budget by providing its size in number of megabytes. This way, we see teams setting up a cache strategy per user-configurable graphics quality level they offer in their game, for example, Funcom as we previously mentioned. We also see teams sometimes set up a dedicated cache for their Hero asset.

You can set the maximum amount of tiles that can be uploaded each frame, pin down certain tiles in the cache, and determine what happens when the cache starts thrashing. You can configure which part of the cache is reserved for previously seen and newly seen textures. For example, a very linear game can have a caching strategy that very quickly evoke previously-seen unused tiles. You can also set a MIP bias, a bias that automatically lowers quality in the scene when the quality is not needed (e.g., fast moving camera with motion blur) or the streaming engine starts lagging behind. Once it recuperates, the bias gets lowered and the quality gradually starts to increase. Finally, you can control the level of anisotropic filtering and enable/disable trilinear filtering.

Tools Pipeline Integration

The Granite SDK tools are designed to be easily integrated into your existing production pipeline. The tools can be integrated in such a way that they have no impact on the artist integration time in your engine. In fact, by using our proprietary compression format instead of BC/DXT, assets can even be encoded and presented faster in the engine.

Granite SDK comes with two tools for creating and managing Tile Sets: a command-line interface (CLI) and a GUI-capable tool. The CLI tool, GrBuild, can be easily steered from existing tool pipelines through shell scripts. A typical run of the CLI tool starts with importing existing texture images and combining them into Stacked Textures (a group of textures stacked together belonging to the same mesh texture coordinates in a single logical unit). Figure 8 shows the entire build pipeline.

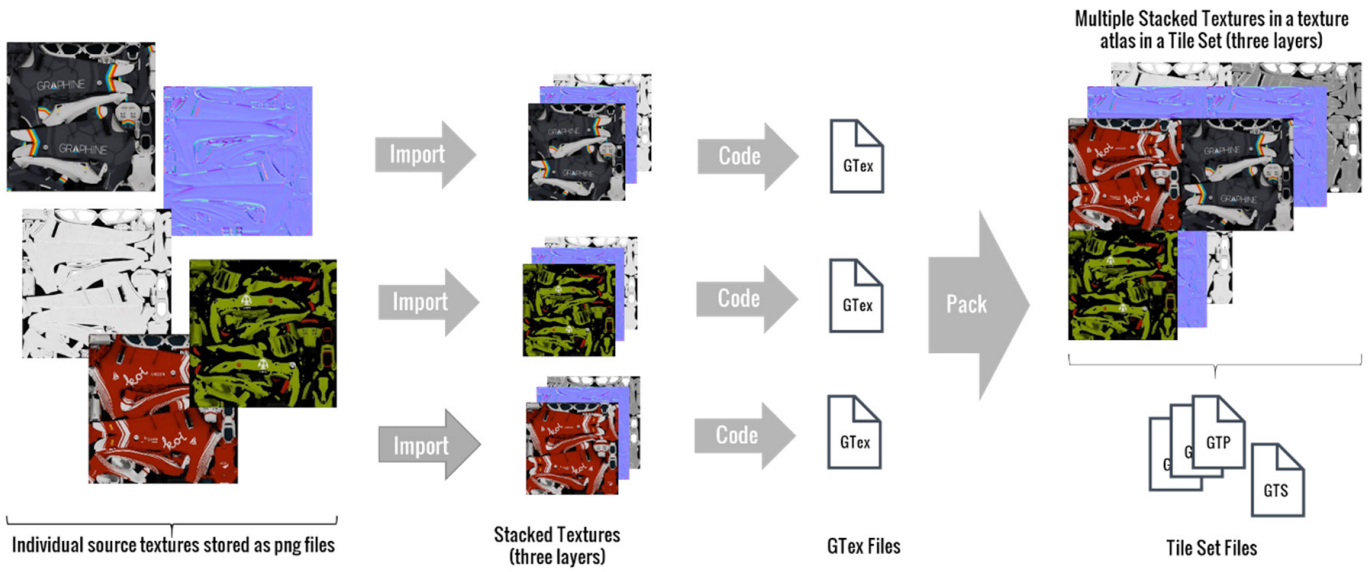

Figure 8: Import pipeline: from individual images to Stacked Textures, GTex files, and atlased Tile Set files.

```
<?xml version="1.0" encoding="utf-8"?>
<AssetImport>
    <LayerConfig>
        <LayerDescription CompressionFormat="Granite" QualityProfile="High" DataType="R8G8B8A8_SRGB" DefaultColor="#FF69B400"/>
        <LayerDescription CompressionFormat="BC5" QualityProfile="Default" DataType="X8Y8Z0_TANGENT"/>
    </LayerConfig>

    <Imports>
        <Import Name="MyAsset">
            <LayerTexture Src="./assets/texture_d.png" QualityProfile="Low" Flip="Horizontal"/>
            <LayerTexture Src="./assets/texture_n.png"/>
        </Import>
    </Imports>

    <ImportTemplates>
        <ImportTemplate Name="*">
            <LayerTextureTemplate Src="./diffuse/*"/>
            <LayerTextureTemplate Src="./normal/*" Swizzle="3120" Flip="Vertical"/>
        </ImportTemplate>
    </ImportTemplates>
</AssetImport>
```

Figure 9: Example of XML import script.

One of the basic operating mechanisms to manage Tile Sets are import scripts. These XML scripts describe the Stacked Textures in the Tile Set, how they need to be imported, coded, and build. Figure 9 shows this. The XML describes a path to the textures, and any basic image operations such as resizing, channel inversion that the tool supports. The import script also sets the compression format, ranging from our high-compression proprietary coding format to GPU-compatible formats on disk (e.g., BC5 on disk). An import script is typically used for the CLI interface, evidently, the GUI does not require these scripts but instead allows the user to import using Graphical User Elements.

The import pipeline is flexible. Some users and teams set up an import script for each individual stacked texture. Others set up a single import script that explicitly enumerates all their stacked textures. Yet another option is to let our CLI tool scan a directory for textures and set up stacked textures automatically according to an import template.

Next in the pipeline, a Stacked Texture is coded to a GTex file, a Granite Texture (see again Figure 8). This file contains all tiles for the entire Stacked Texture, including its mips, in a coded format. Conceptually, it is comparable to a DDS texture of your original source texture. The Granite runtime can stream from these GTex files, but only during development. GTex files are built once and shared during production. Hence, they are optimized to a lesser extent for streaming and more for portability and sharing during production. For best streaming results, an optimized Tile Set file needs to be generated based on these GTex files. This is conceptually comparable with packing DDS files (the GTex's) into a large container format like a multi-file zip (the Tile Set files).

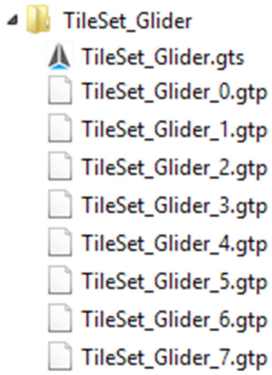

Figure 10: Example of a GTS accompanied by GTP files.

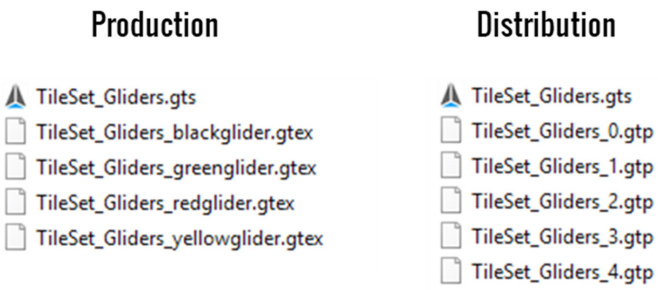

Figure 11: GTS files accompanied by GTex (production) or GTP (distribution) files.

When generated, a Tile Set consists of a GTS Header file and accompanying GTP Page files. Figure 10 illustrates this. As the name suggests, the GTS header file does not contain any tiles, instead, the accompanying GTP page files do. These page files are built to achieve optimal streaming performance. They cluster tiles in pages, a block of data aligned for optimal IO reading requests, typically sized to 1MB. These pages are then reorganized in special access pattern order, and have any duplicate data removed. GTP page files can be optimized per platform, although we see many users and teams share their Tile Set files across platforms, some even across PC and mobile platforms.

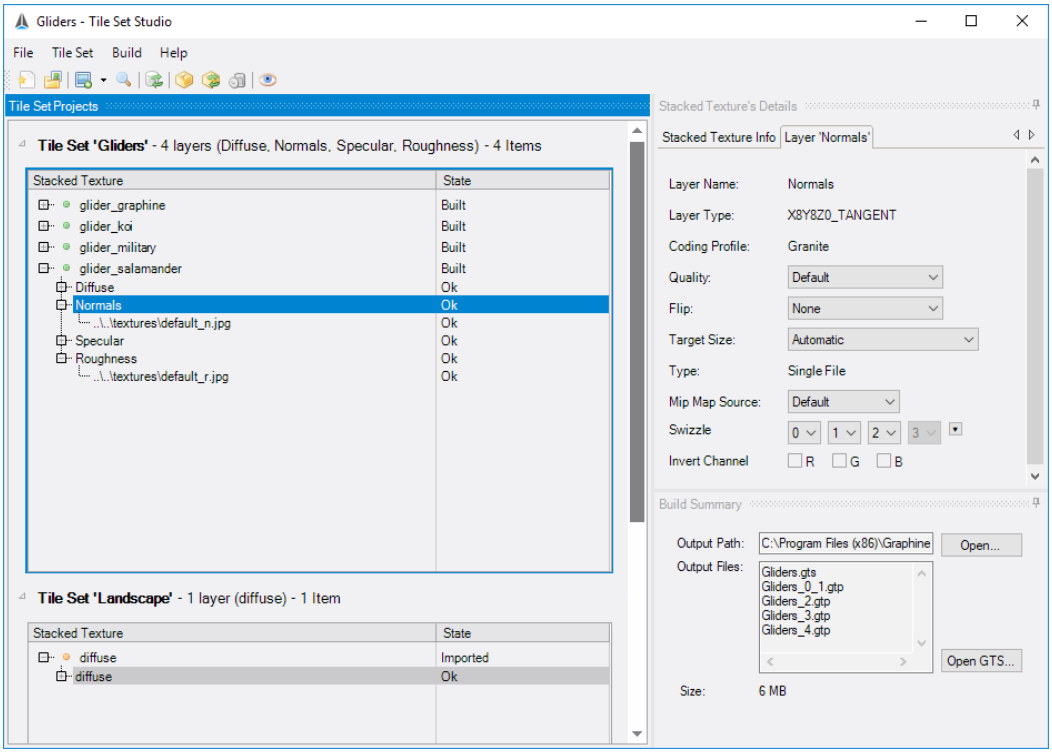

Figure 12: Tile Set Studio, a GUI application for easily creating and managing Tile Sets.

During production, you can choose to stream from GTex files or from GTP Page files. Because GTex files lack the aforementioned optimizations, they take less time to generate than GTP page files, and they are more self-contained to be easily sharable in teams. GTP page files on the other hand take longer to generate because they are optimal towards streaming for a specific platform. Figure 11 illustrates the files for streaming during production (left) and distribution (right).

Finally, the GUI tool, Tile Set Studio, has the same functionality as the CLI tool but with a Graphical User Interface, see Figure 12. We typically see users and teams of large productions use the GUI tool to set up the initial CLI import scripts and to familiarize themselves with the capabilities of the tools.

Tile Set Distribution and Patching

When building a Tile Set, the Granite tools produce a number of different files: a GTex file for each individual Stacked Texture, and one GTS header file accompanied by multiple GTP page files, holding all your Stacked Textures. As mentioned before, GTex files are intermediary files that are shared and streamed from during production. The GTS header and its GTP page files on the other hand are the files that get shipped with the final build of the application. They are built for optimal streaming for the final build, on one or more target platforms. Copy the GTS and GTP files to your application's content folder, point the Granite SDK to the GTS header file, and Granite can start streaming.

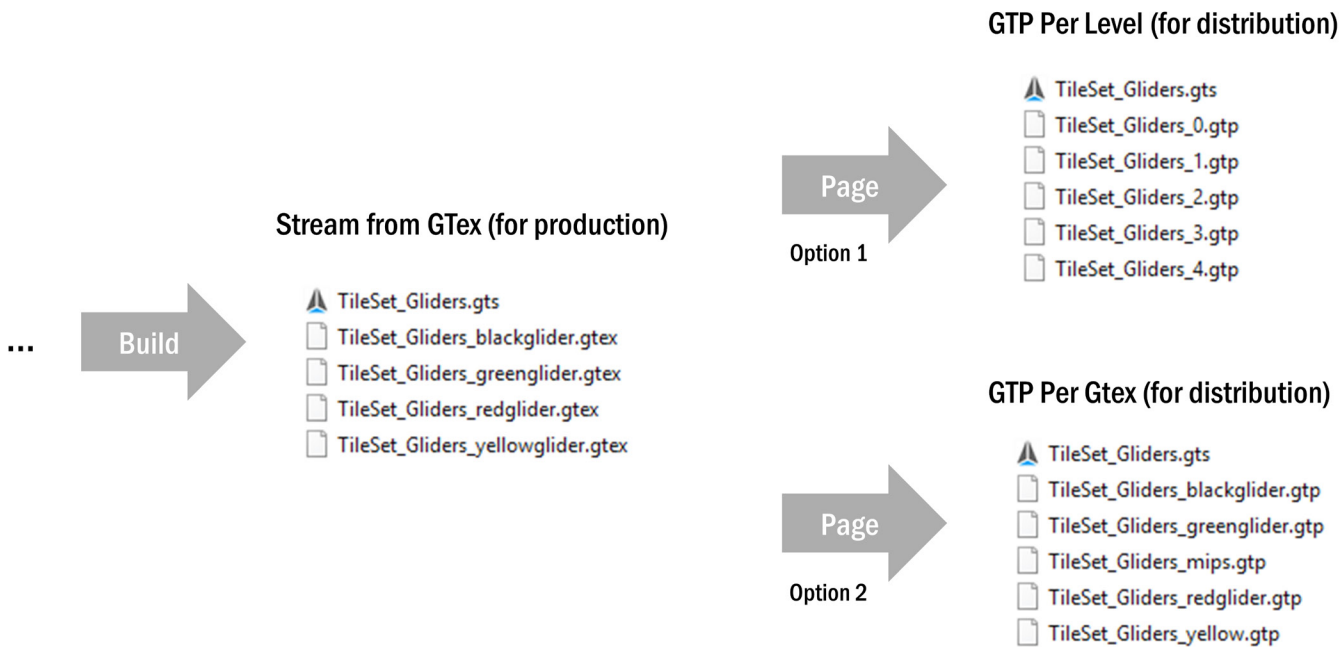

Figure 13: Examples of two GTP paging schemes and the files generated by.
Note the Streaming from GTex scheme available during production.

A Tile Set typically consists of multiple GTP page files. One of the reasons for this is it to make distribution and any future content patching easier. Instead of distributing one very large file (e.g., 2GB), you distribute a number of smaller files (e.g., 20 files of 100MB). Any changes made to the Tile Set in a later patch, makes more granular and therefore more manageable changes to the GTP files. For example, instead of building and redistributing a 2GB changed file, you now build and redistribute only one out of twenty 100MB files.

You can choose different GTP packing strategies according to your needs. The Granite SDK is flexible in streaming whichever data is in a GTP file. Currently the tools offer two strategies, which Figure 13 illustrates:

- One GTP page file per mip map level. You get a small number of relatively large files, a strategy best suited for optimal streaming performance.
- One GTP per Stacked Texture (in other words, one per GTex). You get a larger number of relatively small files, a strategy best suited for patching. Such a strategy tries to minimize the number of changes in multiple iterations of GTP files over the life time of a product. Indeed, when one tile is changed, only a small number of small-sized GTPs need to be updated.

During development, teams can stream from GTex files directly. Any changes to a Stacked Texture result in a single GTex file being updated, so only this file needs to be pushed to version control or any sharing platform. Finally, Figure 14 summarizes which Tile Set files get introduced and changed when Stacked Textures in an existing build are added or updated.

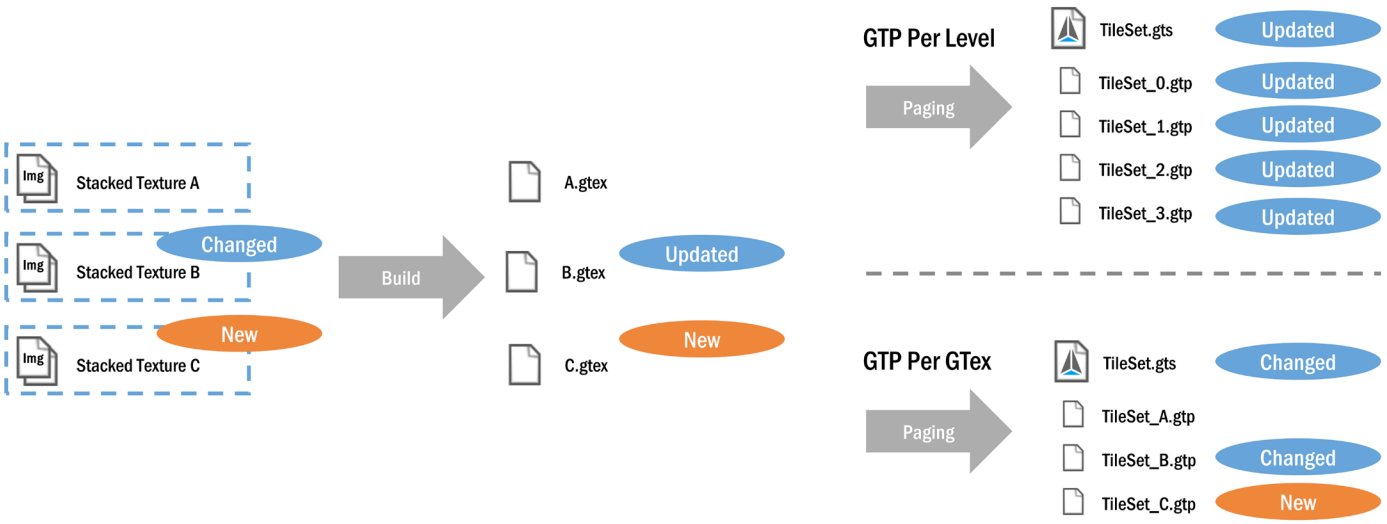

Figure 14: Effect of two paging strategies after updating and adding Stacked Textures in an existing build. On the top, GTP per Level, on the bottom, GTP per GTex.

Version Control and Team Work Flow

Granite SDK 5 works well with version control systems such as Git and Perforce. Figure 15 summarizes the files typically checked in into such a system. In the usual scenario, these are the project files, source images, and GTex files.

We see teams using different types of collaboration work flows with Granite textures. Some teams build both GTex files and Tile Set files on their local system, and share these, either through version control, or over their favorite file sharing platform. Sometimes there is only one artist responsible for Granite-enabled materials, sometimes all the artists build their Tile Sets. For example, an artist changes a material, which changes the corresponding Stacked Texture. He builds the GTex and Tile Set files on his local system, and checks in these files in Perforce. His team members check out the updates files and use them in their project.

This works for relatively small Tile Sets that build relatively fast. As soon as the Tile Set gets larger, it is beneficial to build only the GTex files locally, and stream from these GTex files during production. For larger Tile Sets, teams sometimes set up a build machine to build the Tile Set (See Funcom's blogpost on this here). Daily builds can then stream from GTex files. For example, an artist changes a material, he then builds the corresponding GTex on his local system, and checks it in in Perforce. He streams from these GTex files. His team members check out the updated GTex and also stream from GTex files in their project. Every time a testing build of the game is made, Tile Sets are built on a dedicated build machine and included in the testing build.

If during production you want to share the optimized Tile Set files instead of the GTex files, you need to put careful thought into which GTP generation strategy you choose. The GTP-per-Level strategy will result in a number of potential large files that need updating even if only one tile has changed, as Figure 14 showed.

The GTP-per-GTex strategy will generate a large number of relatively small GTP files. In this case, touching one tile will result in a small number of small files being changed.

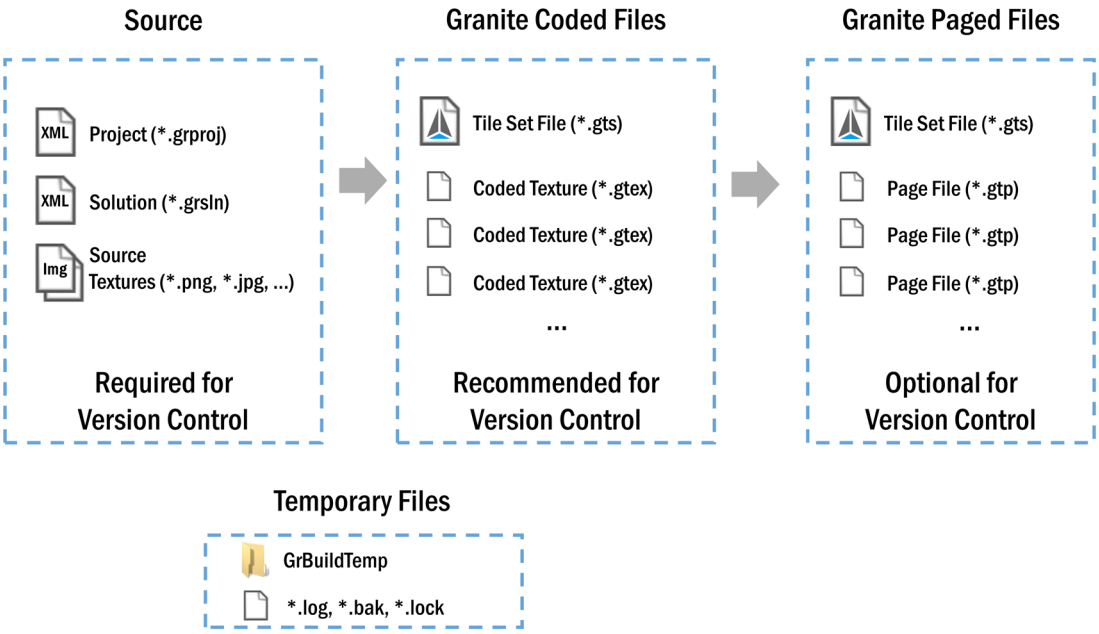

Figure 15: Overview of which files to put under version control

Note that we recommend keeping the original source files. Managing GTex files conceptually compares to managing DDS files. A GTex file represents a coded texture and if you want to change some properties such as the coding format, you need to rebuild it and therefore have access to the original source images.

Deep Integration

For popular 3rd party engines, Graphine offers a pre-integrated version of Granite SDK. These integrations are production ready from the start. Using Granite is almost as easy as just switching on one flag. The Granite tool will then automatically take care of importing textures in our tiled streamable format, and adjust materials for rendering from it. Most notably we have integrations available for Unreal Engine, and Unity. These integrations show what a deep integration of Granite SDK can look like in your engine.

Our Unreal Engine 4 integration introduces a new Unreal Blueprint node to the material editor called a GraniteStreamNode Node (see Figure 16). To stream a UE4 texture through Granite, the only step to take is to replace the original texture sample node with the GraniteStreamNode node. The integration automatically picks up all UE4 textures behind each GraniteStreamNode , builds the corresponding Tile Set and updates the material to use the Granite sampling methods. The integration enables streaming in the editor and automatically adds all Tile Set files to the final game build. More information on the UE4 integration can be found here: http://graphinesoftware.com/granite-unreal-video-tutorials.

Our Unity integration works a bit different. Here, to stream a material using Granite, the artist converts the material in the editor by changing the shader to a Granite equivalent shader. Default Unity shaders have Granite-equivalent shaders out of the box. Custom shaders need to be adjusted to use the HLSL Granite sampling instructions. Next, you create a Tile Set object, specify its layout and assign it to that material. Behind the scenes, our integration scans for all materials that use the Granite shader, steers the tools to add the material's textures to the assigned Tile Set, and updates the material to use Granite streaming. Building the Tile Sets through a Unity menu option enables streaming in the editor, and packaging and building the final game automatically sets up the final build to stream with Granite. More information on the Unity integration can be found here: http://graphinesoftware.com/granite-unity-video-tutorials.

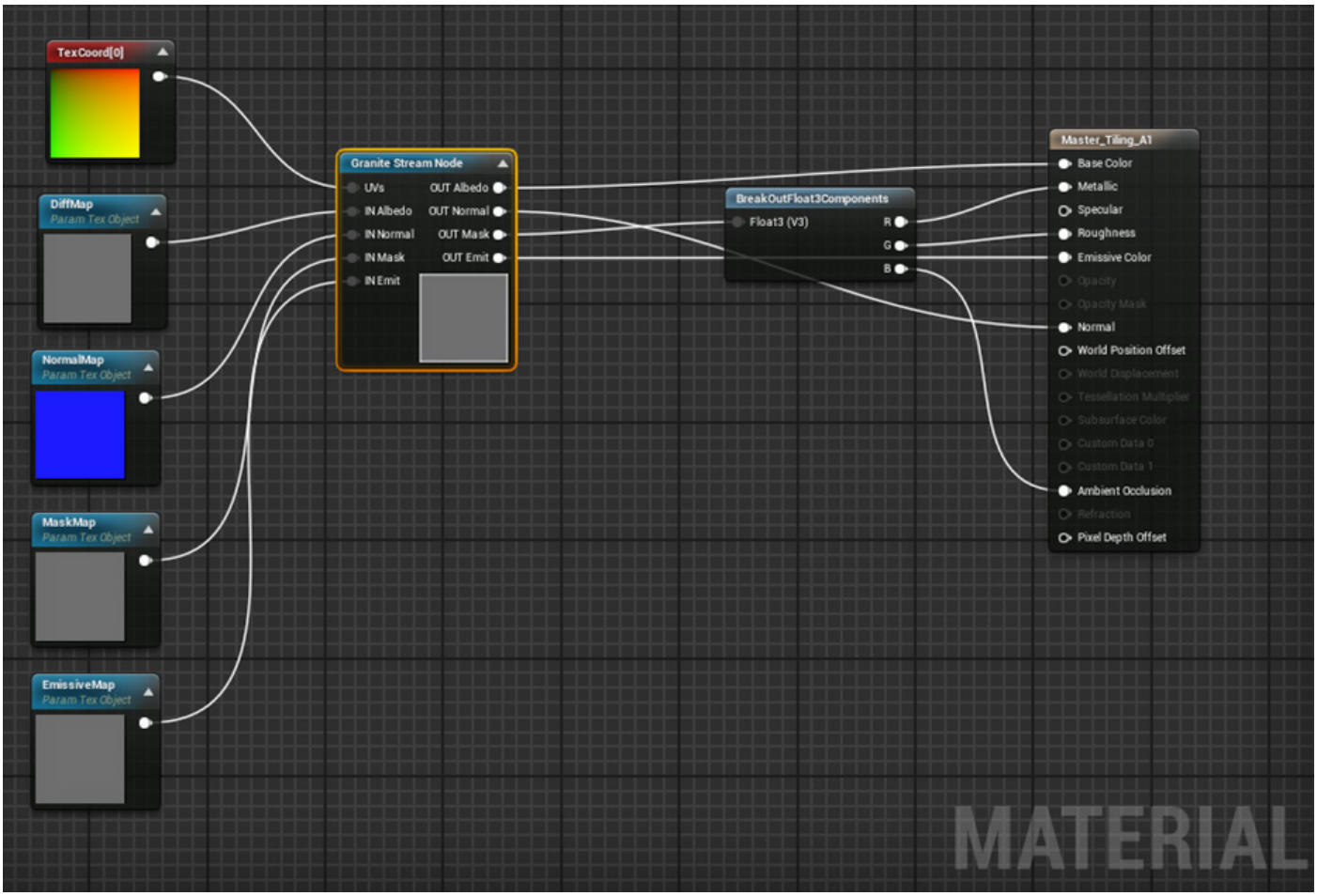

Figure 16: The deep integration of Granite SDK in UE4 puts itself in the material editor where connecting a 'Granite Stream Node' is all it takes to set up materials to use Granite.

System Requirements and Overhead

The minimal requirements for the target system (the End-user) on PC are very specific for the application scenario. CPU and memory requirements for example depend on the type of application, the platform, etc. On the PC platform, a minimal system can consist of:

- Core 2 Duo E6600 2.4GHz or Athlon 64 X2 Dual Core 4800+ CPU
- 3D Graphics card according to tile backend

    - DirectX 11 feature level 10_0 compatible graphics card, (e.g. GeForce 8800 GT, Radeon HD 4850 or equivalent hardware from the same generation)
    - DirectX 9
    - OpenGL ES 3
    - OpenGL
- 128 MB free RAM (even lower is supported)
- Windows Vista and up (32 & 64 bit)

The minimal requirements for the development system working with Granite SDK are:

- Windows 7, 8, 8.1, or 10 64 bit
- 4 gigabytes of RAM
- DirectX 11 compatible graphics card

Using Granite SDK streaming system does introduce some strain on the target system, but this typically has little effect to the overall performance of the system. In fact, Granite is frequently used in VR scenarios where the budget for additional computations is extremely low (e.g., 90Hz has only 11 milliseconds per frame). On the GPU, the cost Granite introduces comes from running extra shader instructions when sampling Granite textures, and bookkeeping texture tile ids in an additional render target. On the CPU, the cost mainly comes from loading and transcoding tiles. Additionally, reading tiles from hard disk puts some strain on the IO bus. Although the actual performance impact of streaming with Granite SDK depends on the sampling rate of Virtual Textures, the compression settings, the scene, etc., we see teams that use Granite SDK such as Funcom generalizing its performance cost to about one millisecond of CPU time per CPU core and one millisecond of GPU time per second[2].

Other Topics

Avoiding Texture Popping

When a texture is not available in time, texture streaming systems such as Granite SDK typically fall back to a low-resolution version of that texture. This way, there is always something to render on screen. But when that requested higher resolution texture suddenly becomes available, it creates a distinct visual event when the higher resolution texture is rendered. It 'pops' into view. This behavior is typically experienced by the viewer as disturbing and should be avoided.

---

[2] Graphine presented together with Funcom their results and experiences on integrating Granite into Funcom's Conan Exiles at the Nordic Game Conference 2017. The presentation can be found here: https://conf.nordicgame.com/sessions/textures-ultra-masses-conan-exiles .

Teams using Granite SDK report they seldom see popping artefacts with Granite. This is because first, Granite has a much optimized tile loading pipeline. Lots of times, it takes only a very small number of rendered frames (e.g., 16 milliseconds for 60Hz) between a tile's request and the moment it is available. Second, Granite explicitly avoids popping by prefetching tile data in advance, hence, making sure that once needed, the data is already present in the caches. Third, Granite allows the user to explicitly control tile loading. Tile cache misses can be anticipated by loading specific tiles in advance. Tiles can be pinned in memory such that they are never evicted. And a special resolve camera can be used to 'preview' a scene (e.g., put a camera at the other side of a teleportation portal). And last, as previously discussed, a LOD bias mechanism gradually throttles down the streaming once many cache misses start to occur. The mechanism steers Granite in streaming lower resolution tiles for the entire scene over a short period of time. Once lesser cache misses occur, higher resolution tiles start to get loaded in gradually.

Instant Play and Optimizing Loading Times

Texture streaming does not load textures in advance in memory but streams textures gradually into texture caches after which they can be used immediately. As a result, there isn't really a pin-point-able texture loading period at the start of an application. Instead, your application starts immediately. In fact, Granite does not require all Tile Set Page files (GTP files) to be present on disk when streaming. For example, you can start the application when the highest-resolution GTP files are not yet downloaded on disk. Granite will then omit loading any tile located in the missing GTP files. Once the GTP file is present on disk, you can instruct Granite again to load the higher-resolution tiles from the new GTP file.

Optimizing Tile Set Files by Tile Pruning

Granite natively supports sparse Tile Sets, i.e., Tile Sets that had some tiles removed and not stored on disk. Now if we were to cull texture regions never used in the game or cinematic, these regions don't take up storage space and the Tile Set becomes smaller on disk[3]. Example use cases include culling of texture detail not used for cinematics, removing too highly-authored texture detail, e.g., fine-grained details on a mountain texture only viewed at a large virtual distance, removing hidden texture detail. For example, we ran our experimental Tile Set optimizer on Epic's A Boy and His Kite demo. This demo has a two minute long cinematic showing a 100-square-mile landscape with many photogrammetry-captured textured objects. Our optimizer removed all texture detail that isn't shown in the cinematic. From a 4.5GB Tile Set we went to a 230MB Tile Set. We replaced the original tile sets with the culled version, replayed the cinematic, and the output was visually identical.

[3] The idea of tile pruning or visibility-based optimizations was originally presented by J.M.P. Van Waveren for Id Software's game Rage. More information on this topic can be found here: http://www.mrelusive.com/publications/papers/Software-Virtual-Textures.pdf.

Streaming Tile Sets over a Network and Visualizing Tile Sets in WebGL

One experimental feature based on Granite 5 our team is working on is streaming tiles directly over a network such as the internet. Instead of loading tiles from disk or generating them on the fly, they are requested from a network server such as a web server and streamed over the network. This allows you to start an application almost instantly without having to download any texture data beforehand. Textures are streamed in over the network the moment they are required. We also have an experimental WebGL Runtime component capable of streaming and visualizing tiles directly in the browser using the WebGL API (Figure 16).

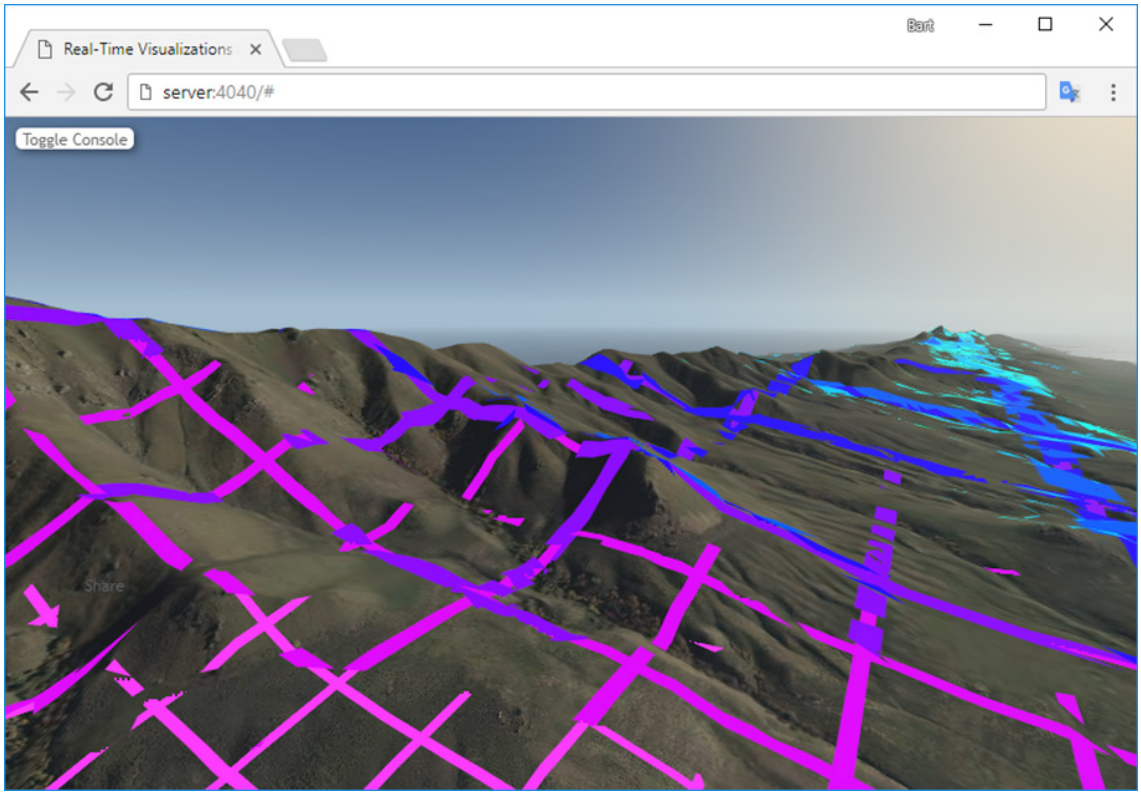

Figure 16: Screenshot of experimental WebGL streaming in Google Chrome visualizing a landscape textured with a 122,880 x 122,880 pixels large texture.

In Summary

Granite SDK streams your textures, no matter which type of texture, how many you have or how large they are, or which type of game or application you have. The Granite runtime runs on millions of devices worldwide on many platforms. Its production pipeline is industry proven by some of the leading game production studios such as Wargaming, Sumo Digital, and Funcom. Granite is continuously being developed by a dedicated team, which is available for working closely with you to reach your goals. Custom features can be added on requests. If you are interested in Granite SDK, or would like to receive more information, please contact us info@graphinesoftware.com.