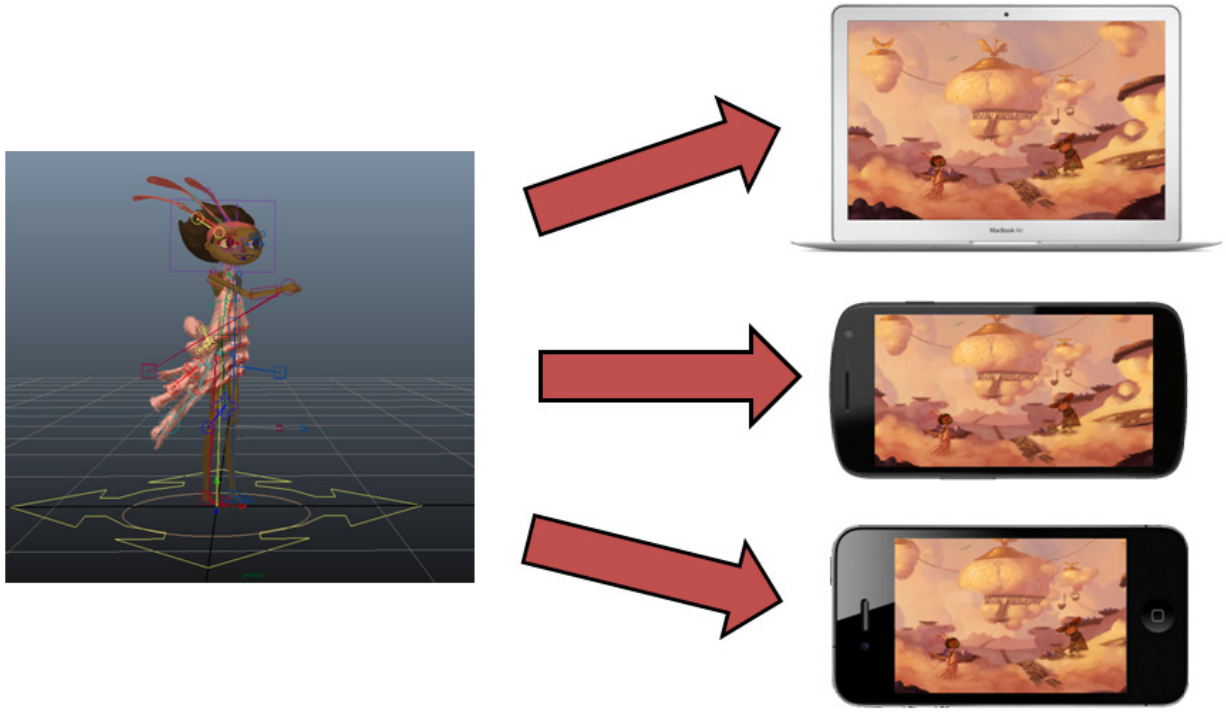
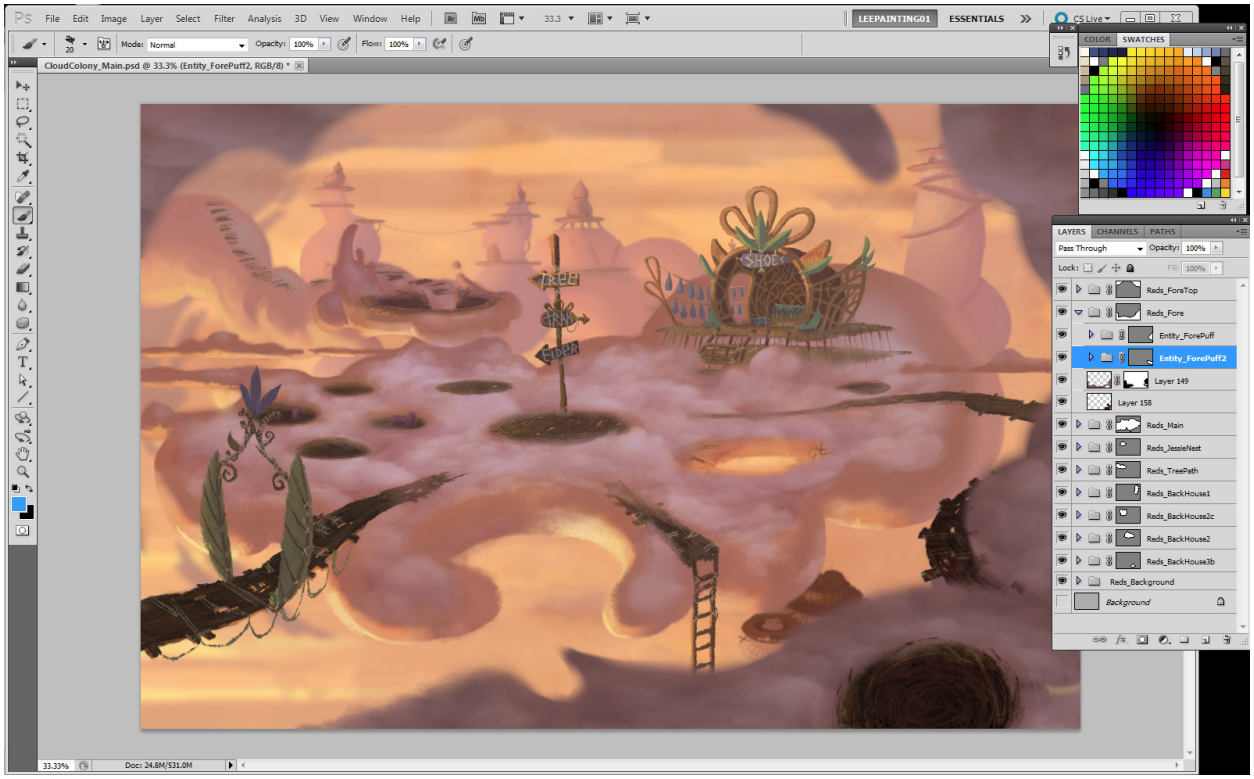
Note: This article was originally released on the Broken Age backer forum as a technical post. Please note that you can still become a slacker backer, which will not only get you the game and the documentary by 2 Play Productions but you'll also get access to more posts like these detailing the making of Broken Age: http://www.doublefine.com/dfa

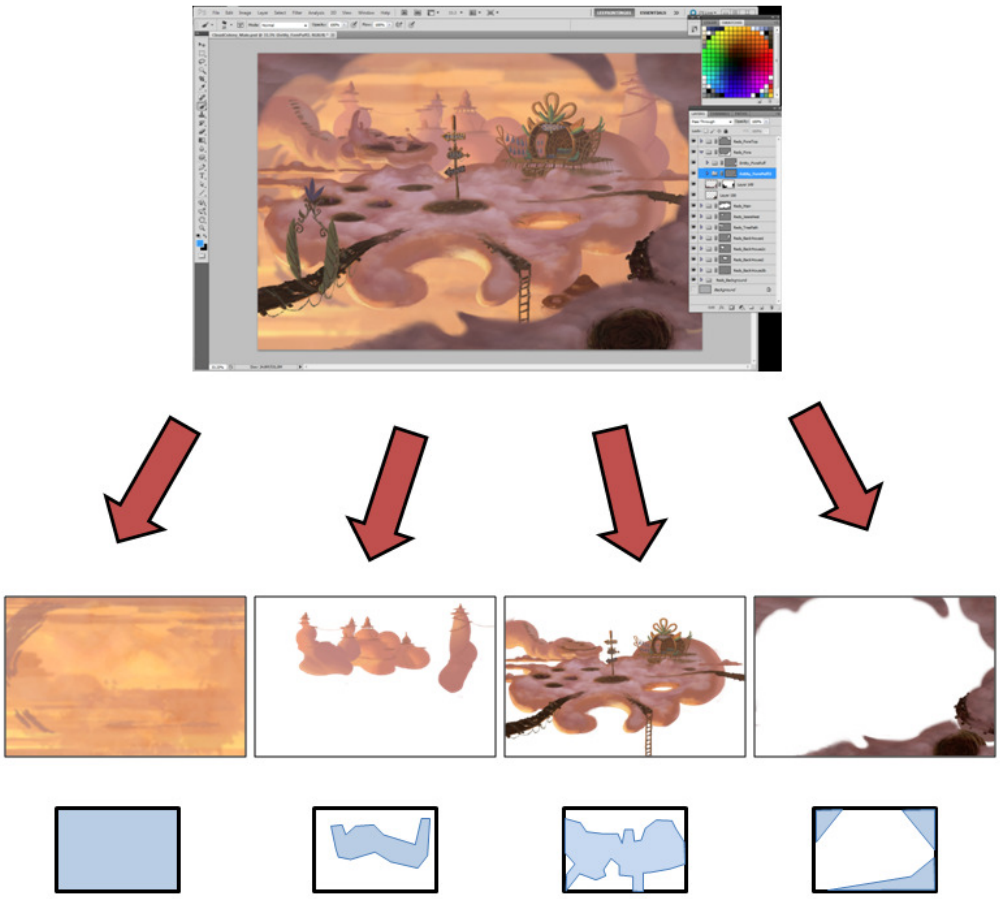
Today I want to tell you guys about a EXTREMELY important part of game development that is very often overlooked, because it's not as glamorous as game-play programming or writing an engine: The data pipeline! In a nutshell the data pipeline is responsible to prepare the assets created by our awesome artists to run optimally on the different hardware platforms. It's actually quite an interesting problem and hopefully I can show you that there is way more stuff happening than you might think.



But why do you even need a step between the art creation tools and the game? While it is true that one could load the raw data files directly into the game it is very often not desirable mostly for memory and performance reasons. Let's look at image assets as an example, because it is the most platform-specific data type for us right now.



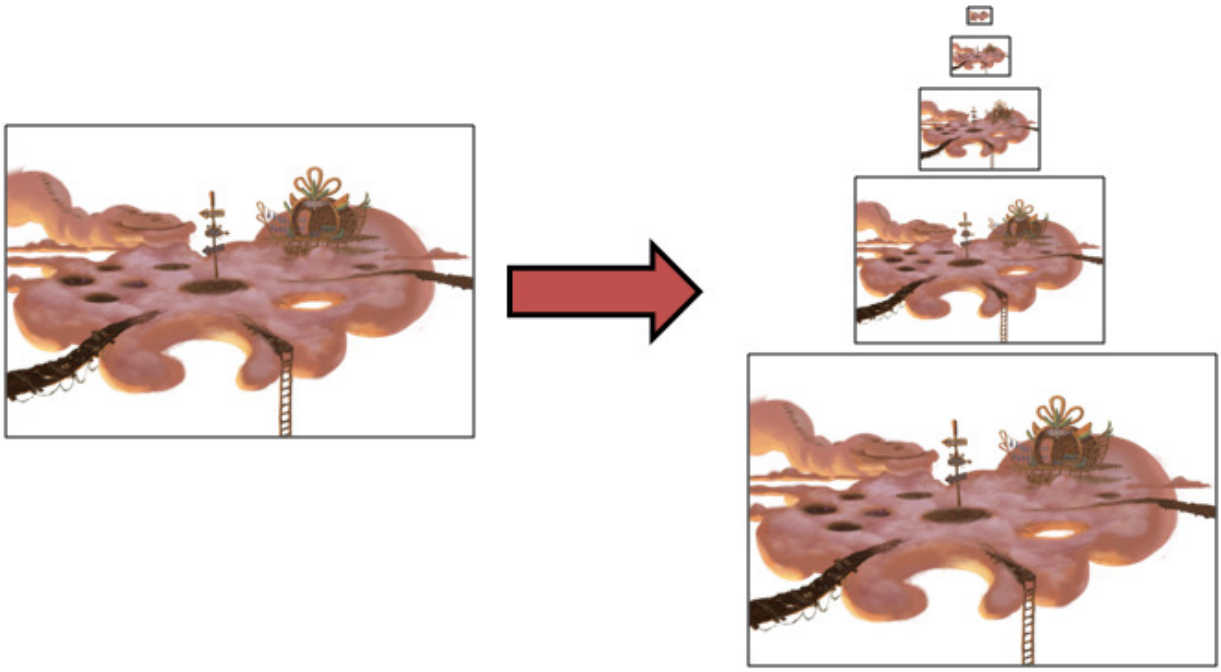
As Lee described in a [previous forum post] the artists are using Adobe Photoshop to paint the scenes as a multilayered image which then gets saved as a PSD file (Photoshop's native file format). In order to see the scene in the game they have to export the relevant image data. This is done by running an export script that will write out the individual layers as PNG files with associated clip-masks. Let's ignore the clip-masks for now so that we can concentrate on the image data. This export script is the first step of the data pipeline for images and its main benefit is that it automates a lot of (tedious) work that the artists would otherwise need to do manually. It is a general rule of thumb that a manual workflow full of monotonous steps is a huge source for errors, so it is almost always a good idea to automate the work as much as possible.



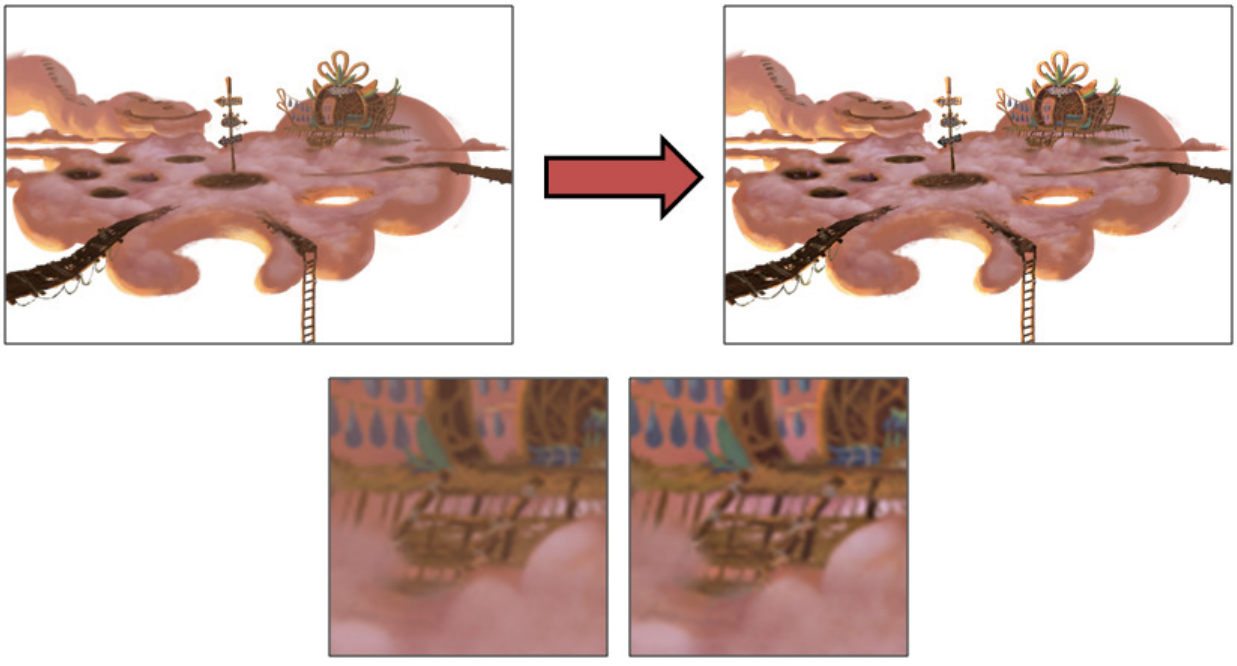
Now since we converted the scene from a high-level asset into individual layer PNG files (and clip-masks) we can start to get the separate images ready for the specific GPUs it'll be used on. The second step of the data pipeline is actually quite complex and contains multiple smaller steps:

1) Mip-map generation
2) Mip-map sharpening
3) Image chunking (only for scene layer images)
4) Texture compression

Mip-map generation takes the image data from the individual PNG files and generates successively smaller versions of it. This is called an image pyramid or mip-map chain. This is important because the graphics chip automatically uses these smaller versions to efficiently draw distant or small objects without visual noise (caused by aliasing).

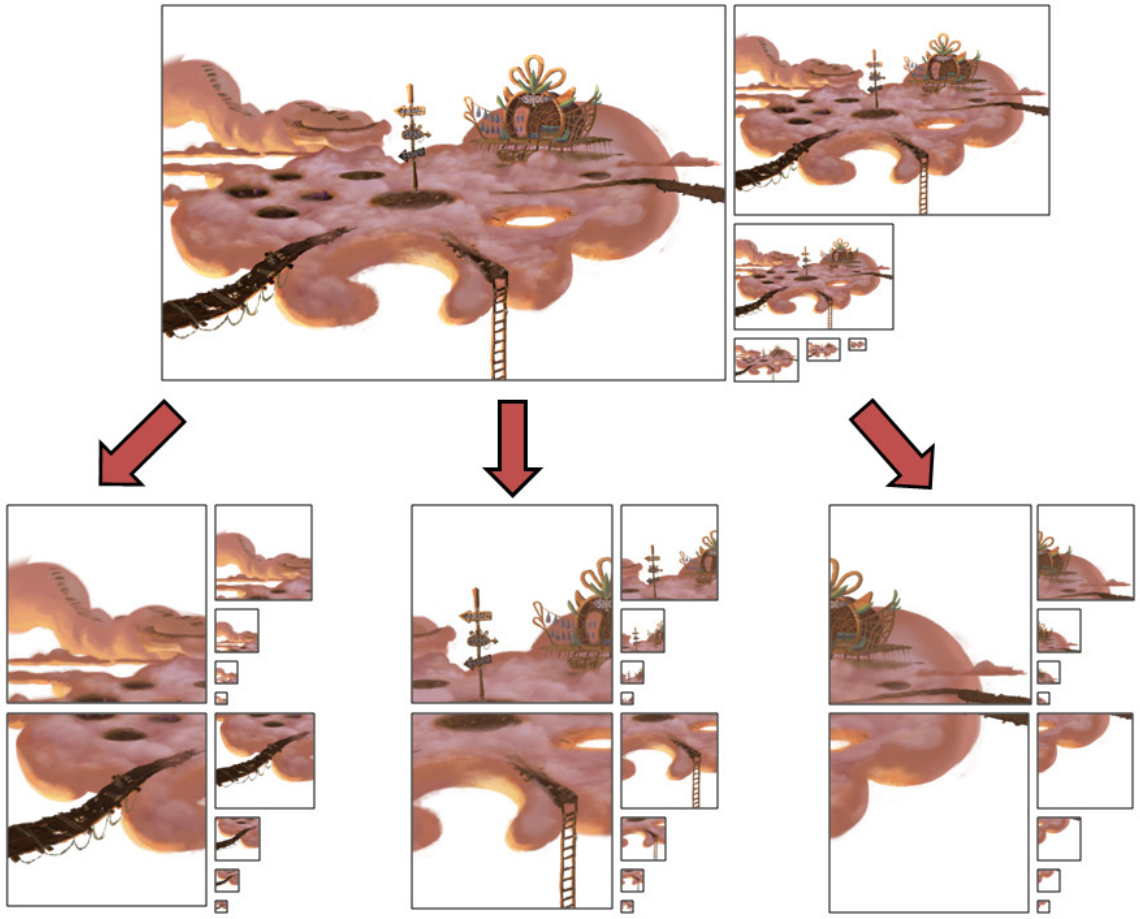


Reducing the size of an image basically means that pixel colors are averaged (or blurred). Unfortunately this very often reduces the contrast in the smaller images. In order to counter this problem the second sub-step increases the local contrast of the mip-maps using an image sharpening filter.



The third sub-step is called chunking. Image chunking deals with the fact that GPUs prefer textures that have power-of-two resolutions. In fact some graphics chips require the textures to also be square-shaped. It is impractical however for our artists to draw the scenes with these constraints in mind, so

this pipeline sub-step splits the large and irregularly shaped images into smaller (square) textures with power-of-two resolutions. The appendix at the end of this forum post will describe in greater detail why GPUs prefer textures with these constraints.



The fourth sub-step converts all the image chunks into the hardware-specific data formats. In order to support all the hardware platforms we are committed to it is necessary to convert the chunks into 4 (!) different texture formats: DXT (Windows, OSX, Linux, Android), PVR (iOS, Android), ATC (Android) and ETC1 (Android). These formats have different requirements and characteristics which actually had quite a big impact on the engine. If you are interested in why we are using these texture formats rather than loading PNG images directly into the game you can check out the appendix at the end of this forum post. Be warned though it is quite technical.

At this point the images are basically ready to be used in the game. Depending on the type of asset or the target platform there might be other pipeline steps though (e.g. file compression using gzip).

Here in Double Fine we call the second pipeline step “munging”. Other names for this process are “data cooking”, “content processing” or “data builds”. Here is a list with some of the asset types we use with their associated data pipelines:

- Images (character textures, scene layers, other textures)
    1. Export from Photoshop
    2. Munging
        a. Mip-map generation
        b. Mip-map sharpening
        c. Chunking (scene layers only)
        d. Texture compression
    3. File compression (iOS PVR files only)
- Character models
    1. Export from Maya
        a. Extract hierarchy of joint-transforms
        b. Extract meshes and group them by materials
        c. Calculate normalized skin-weights for all vertices
    2. Munging
        a. Count number of active joints per subset
- Animations
    1. Export from Maya
        a. Extract joint-transformation for each frame
        b. Extract subset visibility for each frame
    2. Munging
        a. Strip constant animation tracks in rest position
        b. Strip delta-trans transformations (for non-cutscene animations)
        c. Remove redundant key-frames
- Shaders
    1. Munging
        a. Resolve file includes
        b. Generate shader permutations
        c. Optimize shaders for target GPU (e.g. standard OpenGL vs. OpenGL ES 2.0)
        d. Identify and remove redundant shaders
- Sequences (cutscenes, visual effects, animation events)
    1. Export sequence from sequence editor
    2. Munging
        a. Group commands into sections
        b. Sort commands based on execution priority
        c. Remove redundant data (e.g. fields with default values)

This concludes this forum post about the pipelines we are using in order to get the data ready for the many different platforms the game will eventually run on. I hope I could show you guys that there is actually a lot of work that has to be done to an asset before it shows up in the game. It is a very important part of game development though, because the representation of the data will very often have a profound impact on the memory footprint and run-time performance of the game, so getting it into the optimal format is super critical.

As usual please feel free to ask questions. Also make sure to check out the appendix for the gory technical details of efficient texture representations.

Technical appendix: Optimal texture representation

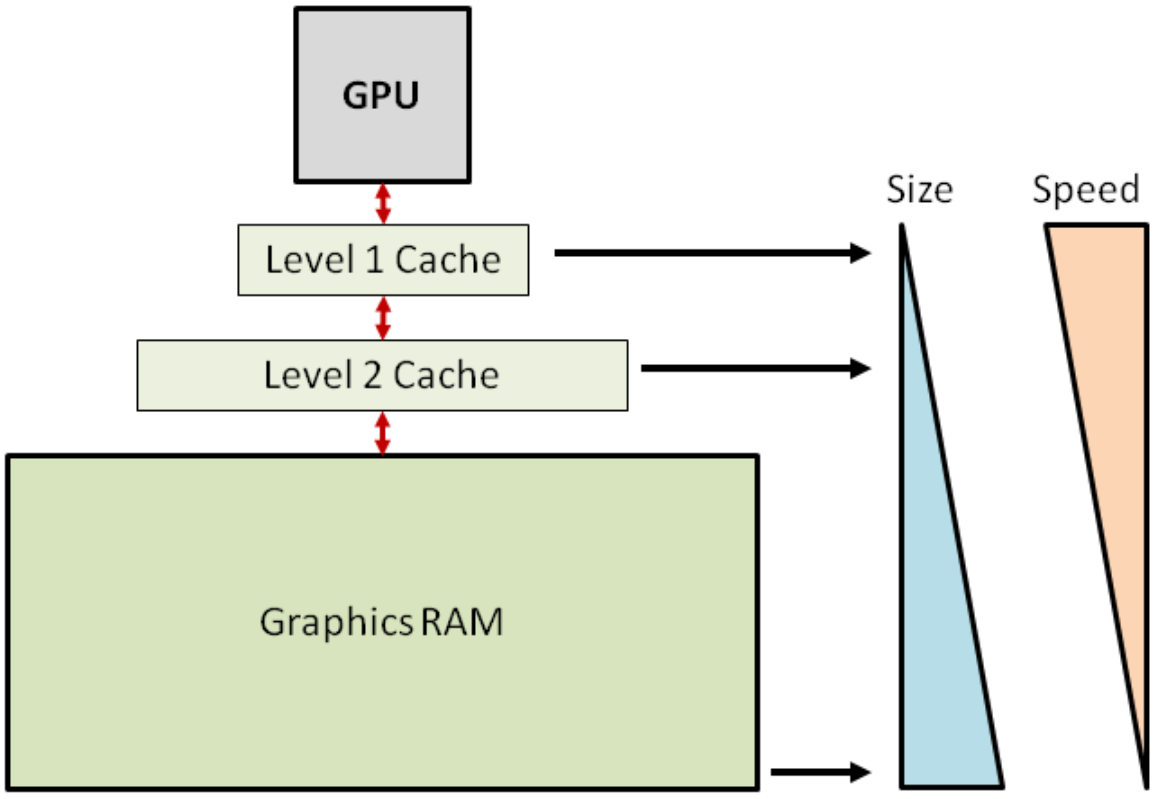
This appendix will describe why games generally don't use standard image formats like PNG, JPG or TGA to store textures. I will also talk about why GPUs tend to prefer square images with power-of-two (POT) resolutions (2^n => 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 …)

It all comes down to data-access speed. While the execution speed of GPUs (and CPUs) has steadily increased both by fitting more transistors onto a the chip as well as adding multiple execution cores the latency to read and write data from memory hasn't improved to the same degree. At the same time modern games require more and more data to achieve a high visual fidelity. Higher screen resolutions require larger textures in order to keep the texel-to-pixel ratio constant. That leaves us in a bad place though because we have super fast GPU cores which need to access more data very quickly in order to render a freme efficiently.

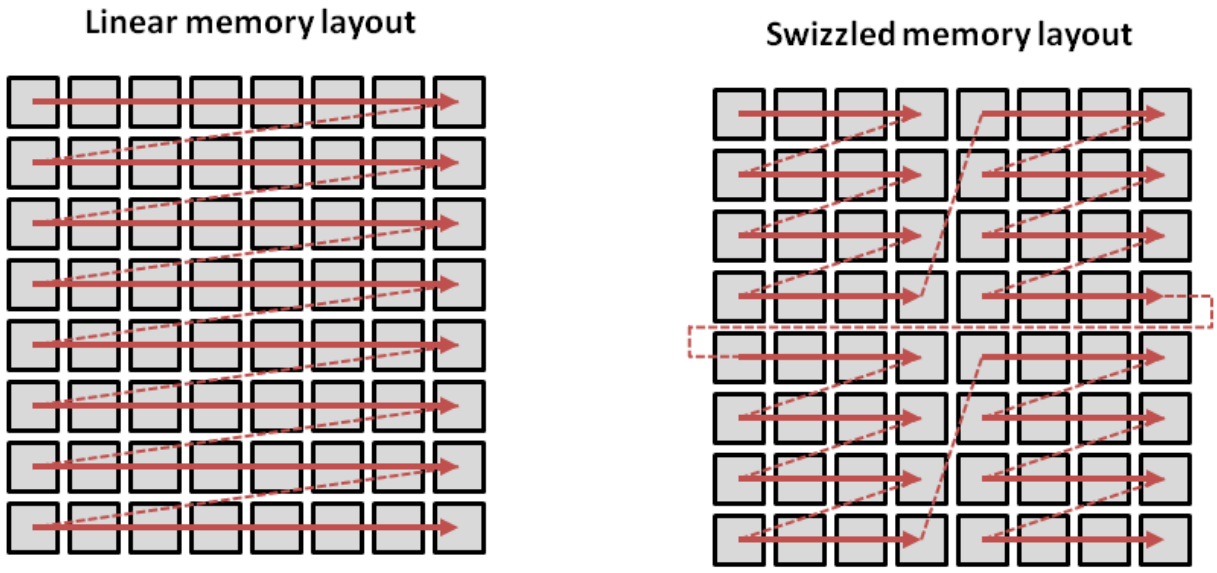
Thankfully there are different things that can be done to improve the situation. As I mentioned above textures are very often represented as an image pyramid rather than a single image. The smaller versions of the image are called mip-maps and require only a quarter of the memory of its parent image. The GPU can leverage the smaller memory footprint of the lower mip-maps for surfaces with smaller screen-space coverage (e.g. objects that are far away or oriented almost perpendicular to the camera).

And this is where it the square and POT requirements come in handy. Textures that abide by these rules simplify a lot of the computations required to look up texture-pixels (called texels) at different mip-map levels. That means that the GPU can find out very quickly what color a pixel should be on screen. There are additional benefits for POT textures too like simplified coordinate wrapping and clamping.

To speed up data access even further the GPU (just like the CPU) uses the benefits of different memory cache levels. The level-1 cache is smallest level in the memory hierarchy, but is the fastest to access. If the processor can't find the requested data in cache it will search the next level which is slightly slower. If the data can't be found in the cache at all a slow non-cache memory access is issued. Rather than just retrieving the requested piece of information additional values are fetched and copied into the memory caches. This is done in order to be able to benefit from data locality. Locality uses the observation that very often when a value is used for computations other data nearby will also be fetched for the following operations. The important thing is that the cache implementation is very low-level and generally doesn't know about the type of data (e.g. vertices, textures), so the memory controller simply copies a linear section of memory into the cache centered on the address that was accessed.



Unfortunately images are rarely accessed in a linear fashion though. For example texture filtering combines multiple adjacent texels into one resulting color. Also graphics chips usually draw 2x2 pixel blocks at a time in order to leverage the parallel nature of the rendering computations. These observations finally lead us to texture compression, because the major goal of this technique is to lay out the texture data in the most efficient way. Rather than expressing an image as a linear array of pixels the data is converted into a block-based (or swizzled) representation. This image shows the difference between the two different data layouts:



In addition to a cache-friendly data layout texture formats like DXT also compress the pixel data by exploiting the fact that the color of neighboring pixels very often doesn't change very much. That means that the difference between adjacent texels can be expressed with fewer bits, which reduces the memory required to represent an image. So the GPU has to deal with less data which is formatted in an optimal way! Hooray!

But that still doesn't explain why we don't use standard image formats like PNG directly though. Well they simply aren't designed to represent images in the optimal fashion described above. Usually these formats don't support multiple surfaces necessary for mip-maps and the image is expressed as an linear array of raw RGBA values. In theory one could load an image from a PNG file and compress it before sending it to graphics memory but the compression requires a lot of CPU and memory overhead. Also this transformation really should only be done once rather than every time an image is loaded.

This leads us back to the data pipeline which is the main topic of this forum post. One of the most important steps of image munging is in fact texture compression, which will convert the raw image data into the data representation preferred by the different GPUs.

I hope you enjoyed this appendix and that I was able to convince you that compressed textures are great and should almost always be used instead of raw images!