Hardware accelerated graphics

Stefan Gustavson (stegu@itn.liu.se) 2015-04-10

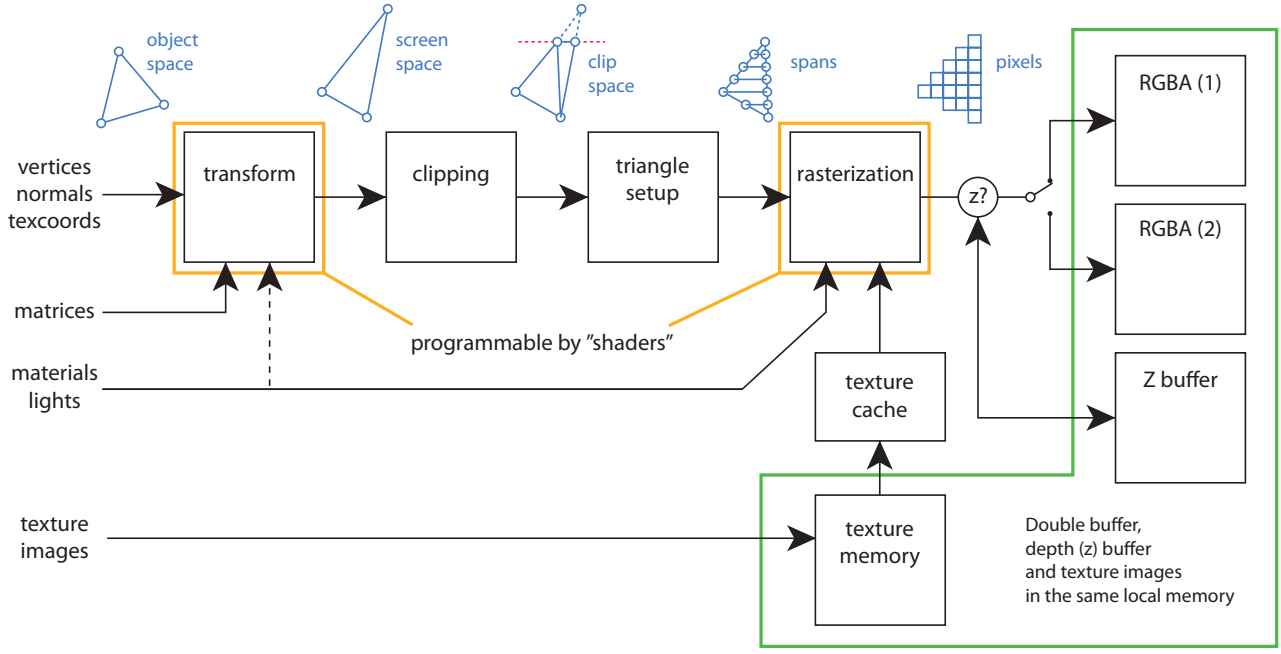
Fifteen years ago, hardware accelerated 3D graphics was an expensive, specialized niche for professional applications, but nowadays it is cheap and ubiquitous. The primary market reason for this development is 3D gaming, but graphics accelerators are very useful also for professional applications, and they present an advantage even for 2D graphics and video. The graphics accelerator is now often the most complex chip in a computer, and it is commonly referred to as a ”graphics processing unit”, or GPU.

Because there is a lot of power to tap into by using a GPU to its full capacity, it is useful to know the general architecture of the pipeline. This makes it more apparent what can be done and, perhaps even more importantly, what *cannot* be done easily within the constraints of the current hardware designs. Good graphics programming requires at least some understanding of what goes on behind the scenes in a modern GPU. Hopefully, this overview will shed some light on that without going into too much detail.

The graphics pipeline

The current generation of GPUs have an architecture that was introduced already in the late 1980's by the company Silicon Graphics (now SGI, no longer doing 3D graphics). While the architecture is old and basically mimics software rendering methods from about 25 years ago, it has scaled well and still seems to have more to offer. The figure below presents the architecture at an overview level. The rest of this document is a detailed explanation of this figure.

The GPU Graphics Pipeline



The primary primitive for hardware accelerated graphics is *triangles*. While recent hardware developments have made it possible to use other representations like volumes, implicit surfaces and parametric surfaces, triangles are still by far the most common representation. Triangle mesh models are typically characterized by *vertex coordinates*, *normal vectors* and *texture coordinates*. 4x4 *matrices* are used to transform the homogeneous vertex coordinates from object space to screen space and to perform a perspective projection. *Materials* and *lights* can be specified for simulation of illumination and reflection, and *texture images* play a very important role in rendering visually complex 3D graphics while keeping the geometry of the scene reasonably simple.

Transformation

The vertex coordinates and normals (and potentially even the texture coordinates) are transformed by multiplication with one or several 4x4 matrices. For the vertex positions, the transformation is followed by a division by the fourth homogeneous coordinate to achieve a perspective projection.

Clipping

After the transformation to screen space, individual triangles can be inside or outside the view, and may possibly cross the screen boundaries. This is resolved by clipping the triangles to remove the parts that extend beyond the screen. Clipping generates some non-triangular polygons that are once again split into triangles for the final rendering steps.

Triangle setup

As a preparation for the final step, triangles are split into one pixel wide line segments, referred to as *spans*. The coordinates for the two endpoints of each span are interpolated between the triangle vertices. Other per-vertex data such as normals and texture coordinates is interpolated as well.

Rasterization

Spans present a fairly simple rendering problem. A loop with linear interpolation between the two endpoints of each span generates individual pixel samples, which are then sent to the framebuffer.

Depth test

Straightforward *z-buffering* is used to resolve occlusions. While this is wasteful in the sense that many pixels that are rendered might be discarded late in the process, it is a simple method which is very easy to implement in hardware. All that is required is that the depth value of each rendered pixel is checked against a previously stored depth value, and allowed to update the frame buffer only if it is closer to the viewer than what has already been drawn in that pixel.

Double buffering

The frame buffer holds the pixel data. In addition to RGB color values, an alpha value (A) is often saved to support transparency and silhouette masks for compositing of several images. To make it possible to display something while an image is being rendered, and to have anything to display even if the rendering of a frame takes longer than the screen refresh rate of typically 60 Hz, a *double buffer* is used. While the latest RGBA image is being displayed, the next frame is rendered, and when the rendering is finished the two are swapped, the previous frame is erased, and rendering of the next frame starts. There is always a fully drawn image in a least one of the two buffers.

Texture memory

Textures play an important part in most 3D graphics applications, and textures are typically applied in the rasterization stage. Many 3D applications use a lot of high resolution textures, and for maximum performance the textures are stored in fast memory local to the GPU. This local memory is shared between the frame buffers (RGBA and Z) and textures, but considerably more space is used by textures. The total amount of memory available to a GPU is often 1GB or more. Just like in a CPU, a memory cache is used to speed up and parallelize the memory access.

Programmability

While the description above is still valid for the typical use of a GPU, there is one more thing to point out. In recent years, GPUs have become programmable at a low level, such that it is no longer a requirement to use only built-in, fixed algorithms for rendering. The transformation and the rasterization steps are now both fully programmable. By using a specialized programming language, the programmer can write small program snippets called *shaders* to take detailed control over what is being done to create and transform triangles and render pixels to the frame buffer. Software shader programming is an old concept in computer graphics, but hardware shaders executing on the GPU were introduced only about a decade ago, and the field is still in rapid development.

Speed

In the early days of graphics acceleration, emphasis was put on relieving the CPU from doing some mundane tasks and balancing the workload between the CPU and a dedicated graphics processing unit. The goal has now shifted towards pushing as much work as possible to the GPU, because it is capable of rendering 3D graphics roughly 100 to 1,000 times faster than a general CPU. The main reasons for its performance can be summarized as *specialization*, *pipelining* and *parallelization*.

Specialization

A GPU is tailored to do graphics, and nothing else. The architecture is simple and does not require the generality and flexibility of a CPU, so everything can be very narrowly focused and optimized for the task at hand. Also, the local GPU memory is faster than the system memory.

Pipelining

The rendering is split up into several separate stages, with each stage handing over its data to the next stage without any need for iterations or backtracking. This makes it possible to have data continuously flowing through the system at all stages. As soon as one stage is done with one chunk of data, it hands the processed data over to the next stage and starts with the next chunk of data right away. This kind of data flow is called *pipelining*, named very appropriately after oil and gas pipelines. You put stuff in at one end and get the same kind of stuff out at the other end at the same rate, but the stuff that comes out was put in quite some time ago. The actual number of pipeline stages is considerably greater than the four steps shown in this simplified presentation, and the data throughput of a GPU is very high because most of its circuits can be kept busy all the time.

Parallelization

Last, and perhaps most importantly, the rendering algorithms for real time 3D graphics are chosen such that each triangle, in fact each pixel, can be processed independently of all others. This makes it comparably easy to have several parallel pipelines, each working on a different set of triangles or a different chunk of pixels, but rendering to the same frame buffer. A high performance GPU of today can have around one thousand identical units working in parallel for some tasks. Even though not all of these can work completely independent of another all the time, there are tremendous gains to be had from parallel computation in 3D graphics rendering, and in recent years considerable effort has been put into removing the performance bottlenecks for this kind of parallel execution.

Limitations

The pipeline architecture presented here is very useful indeed, but it has at least one clear limitation: the rendering is restricted to *local reflection models*. Effects like specular reflections and shadows can be faked by reflection mapping and shadow maps, but other global illumination effects can be very hard to simulate in a convincing manner. Efforts are made to make ray tracing and similar methods viable for real time rendering, but it will be some time yet before true global illumination methods can be used for general, interactive graphics. To some extent, real time 3D graphics is still restricted to rendering methods developed more than 20 years ago. Still, computer graphics has a long history of using successful cheats to get the desired visual result with as little computations as possible, and impressive things can be done even within the restrictions of the current hardware architecture.

GPU computation

Seeing how GPU raw computation performance is now many times higher than that of a general CPU, considerable effort has been spent on making that GPU horsepower available for general computations. Several useful programming interfaces have been developed in recent years to use the GPU for completely different things than to render graphics. Programming efficient algorithms for massively parallel architectures is a complex subject, but it is rapidly becoming very useful.