The GPU

The name *Graphics Processing Unit* has been used to market graphics cards, or on-board integrated hardware of the same purpose. It compliments the *Central Processing Unit* in that it also executes programs and performs computations. However, even if today's CPUs have several cores working in parallel, switching between multiple threads and processes, we are more or less right to think about the CPU as indeed a single unit executing one sequence of program operations at any one time. The GPU, on the other hand, includes several *multiprocessors*, with hundreds, or even thousands of *cores*, executing operations simultaneously. This high parallelism gives GPUs extraordinary processing power, but comes with severe limitations.

Dedicated GPUs have their own memory, called *device memory*, as opposed to *host memory*, which is accessible to the CPU. The cores are able to read from and write to this memory, but that is a slow process compared to the extensively cached memory accesses of the host memory by the CPU. GPUs can make up for this by some highly targeted special caching mechanisms (e.g. texture caching), but mainly by accessing memory in a *coalesced* manner. This means that cores on the same multiprocessor always simultaneously read and write data that is compactly aligned in memory. What is more, these cores share instruction control logic: they must execute the same operation (or leave some cores in the group idle, losing their performance edge). This architecture is only efficient if we have to process large buffers of data, element by element, independently, producing another contiguous buffer of output. Fortunately, this is exactly what we need to perform image synthesis using the *incremental* method, i.e. based on drawing lines and triangles.

The CPU works with the host memory, the GPU with the device memory.

GPUs are efficient when performing the same operation on array elements independently.

Is is possible to view the GPU as a collection of processor cores, running programs not unlike CPU cores, with the above limitations being mere guidelines to achieve high performance. That is what *General Purpose GPU*, or *GPGPU* frameworks like OpenCV or CUDA set out to do. However, GPUs offer specialized functionality for graphics, and an interface presented as a *pipeline* of various data processing *stages*. Some of those stages are in fact implemented

The GPU is a pipeline of processing stages.

by programs running on processing cores, but some are special circuitry for the specific purposes of image synthesis. Graphics libraries like Direct3D or OpenGL (including the embedded systems variant OpenGL ES, and its JavaScript implementation WebGL) operate with such a view of the GPU, and let us set up this pipeline to perform rendering tasks for us. We are allowed to run more limited programs on the processing cores (we call them *shaders*), with pre-determined input and output patterns, which ensures efficient parallelism without having to care about that explicitly.

The purpose of operating the GPU pipeline is rendering, i.e. creating an image. The image is represented as a finite resolution grid of rectangular cells, or *pixels*. Every pixel can have its own color. The image resides in GPU device memory, as a buffer of values specifying pixel colors. This area of memory is thus often referred to as the *color buffer*. The contents of the color buffer are typically presented on the screen of a display device to allow the user to see the image.

The pipeline output is an image in the color buffer.

The heart of the GPU pipeline is the *rasterizer*. This is specialized circuitry on the graphics card, ruthlessly efficient in one task only: rasterizing primitives. *Primitives* are, most often, triangles. Rasterization means drawing them by coloring rectangular cells (or *pixels*) of a finite resolution 2D grid. In fact, the rasterizer is only responsible for identifying the pixels to be colored, and not for actually picking their colors or writing those colors to anywhere in memory. Thus, it can be seen to break down triangles into pixel-sized *fragments*.

The rasterizer processes triangles.

The rasterizer generates fragments.

The functionality of all other stages of the pipeline can be seen as determined by the existence of the rasterizer. Considering the input side, it is triangles that we can drawn efficiently, so everything we need to draw must be somehow represented by triangles. Triangle corners are called *vertices*. The rasterizer needs to know the on-screen positions of the three vertices, so we need a stage to provide those: the *vertex shader*. Considering the output side of the rasterizer, the fragments produced need a color that can be written into pixels of the color buffer. The stage processing the fragments and producing those colors is the *fragment shader*.

The vertex and fragment shader stages are programmable. The programs they can execute are also called *shaders*. They are written in a shader language. Prominent shader languages are actually all very much alike, being based the C programming language, with GLSL (the OpenGL Shading Language) being the preferred choice for OpenGL, and HLSL (High Level Shading Language) for Direct3D. Much of the material of this course deals with what exactly the shaders have to do to compute the on-screen vertex positions or the pixel colors.

We already know that the output of the pipeline is a buffer of data, specifically the color buffer. It is also often called the *target buffer*. The pipeline input is likewise stored in buffers residing in the device memory. Before the operation of the pipeline is started, we will need to fill these buffers with appropriate data. Actually, we may create several buffers to represent several different things we want to draw, and select the appropriate input buffer when drawing a specific *model*. Such a model may consist of some description of shape and color. As the rasterizer is only able to work with triangles, the shapes must be represented by *triangle mesh geometries*. The coordinates describing the positions of triangle corners (i.e. *vertex positions*) are the most essential data that should be given. As the data uploaded obviously describes vertices of the geometry, its storage in memory is called the *vertex buffer*. It is possible to list all vertex positions in the vertex buffer for all triangles, storing three sets of coordinates for every triangle, consecutively. However, as triangles representing a continuous surface often share vertices, this would waste memory, bandwidth, and computational power. If we list identical vertices only once in the vertex buffer, we need a different set of data to indicate which vertex triples span triangles. This array of integer indices is called the *index buffer*, where every triple of integers indicates three vertices in the vertex buffer. The GPU hardware is able to assemble the required input to the rasterizer using this data. Together, the vertex and index buffers define the *geometry*.

Input triangle mesh geometries are specified using vertex and index buffers.

A triangle mesh is often just the best approximation of the actual shape we want to draw. If we would like to render a sphere, but our polygon budget only allows for rendering twenty triangles, we will have to make do with a dodecahedron. But even if rendering just a dodecahedron, we could color the pixels covering its surface as if it was a sphere. This trick, known as *smooth shading*, can drastically improve image quality with low triangle counts. We discuss it in detail when we address shading of 3D surfaces, in chapter **??**. For smooth shading to be possible, we need extra information about the intended surface geometry, not just the vertex positions. Specifically, we need to store the *normal*, i.e. the vector perpendicular to the tangential plane touching the original smooth geometry, at every vertex. This allows us to render both a dodecahedron and a sphere, the vertex positions being identical, but the vertex normals different.

The vertex normals are thus also stored in vertex buffers, alongside the vertex positions. Both are called *vertex attributes*. While positions are practically always present, normals may or may not be required, depending on whether we implement smooth shading in our shader code or not. Thus, the attributes that describe a vertex are not set in stone, but they can be adjusted to our needs. This, as we might expect, requires additional settings telling the GPU how to

Shading normals may be vertex attributes.

interpret the data in our vertex buffers.

Positions and normals, along with the index buffer, define the shape of models. Their apparent color may also be given as a vertex attribute. This works great if there are relatively many vertices and the coloring does not need to have a lot of detail. However, low triangle count geometries can benefit greatly from *texturing*. Texturing means in-memory images can be applied on the triangle surfaces, not unlike decals. This, however, needs a per-vertex specification of what part of the decal image is supposed to end up on which triangle. That can be given in another vertex attribute, called *texture coordinates*. We discuss texturing in detail in chapters **??** and **??**.

Texture coordinates may be vertex attributes.

The same model can be drawn many times. For example, drawing one thousand spheres, at different locations or maybe even different radii, should not require creating and filling a thousand vertex buffers—they should all be drawn using the same model, with the same original vertex positions. Where these vertices end up on screen (i.e. where the rasterizer should draw them), changes when objects move; when they rotate; grow or shrink; but also if our entire view is panned or zoomed. Thus, calculations to this effect, i.e. considering object poses and view settings to find on-screen vertex positions, must be performed for each vertex. This is exactly what the vertex shader units can do. GPU processor cores run vertex shader code, processing all vertex data given in the vertex buffers, in parallel, independently. They write out results into a buffer in memory, from where the rasterizer is able to obtain, using the index buffer, the data for triangle vertices. We discuss how the computations should be performed by the vertex shader in chapters **??** and **??**.

The vertex shader is executed on all model vertices to find their on-screen positions.

Before fragments could be written into the pixels of the target buffer, their colors have to be found. This can be done in many ways, depending on our desired visuals. We write fragment shaders to formulate them. Often texturing and smooth shading techniques are involved in finding the fragment color. In any case, vertex attributes must be taken into account. Note that the rasterizer already needs to process the on-screen vertex positions of the triangle vertices, and produce the in-between positions for all the fragments. This process is known as linear interpolation. In fact, this is the essence of that the rasterizer does: churn out in-between positions for all fragments. This interpolation is extended to every output of the vertex shader. When writing our vertex shader code, we can choose to output any vertex attributes. These will get interpolated, and the intermediate values passed to the fragment shader as inputs.

In the above, we assumed that there is a mechanism to set some parameters and use them in the shaders: e.g. the view settings and object poses in the vertex shader, the texture in the fragment shader.

Note that these are different from the vertex attributes (i.e. the position, normal, or texture coordinates). Attributes have different values for all vertices. The above settings, on the other hand, are the same for all vertices. Therefore, they are called *uniforms*. Uniforms are set from the host program before the operation of the pipeline is started. Shaders can read, but not change them. Before drawing a second object, the host program may set different values for the uniform parameters, which then apply for all the processed vertices again.

Attributes are vertex shader parameters that are different, uniforms are the ones that are the same for all vertices.