M30242 Graphics and Computer Vision

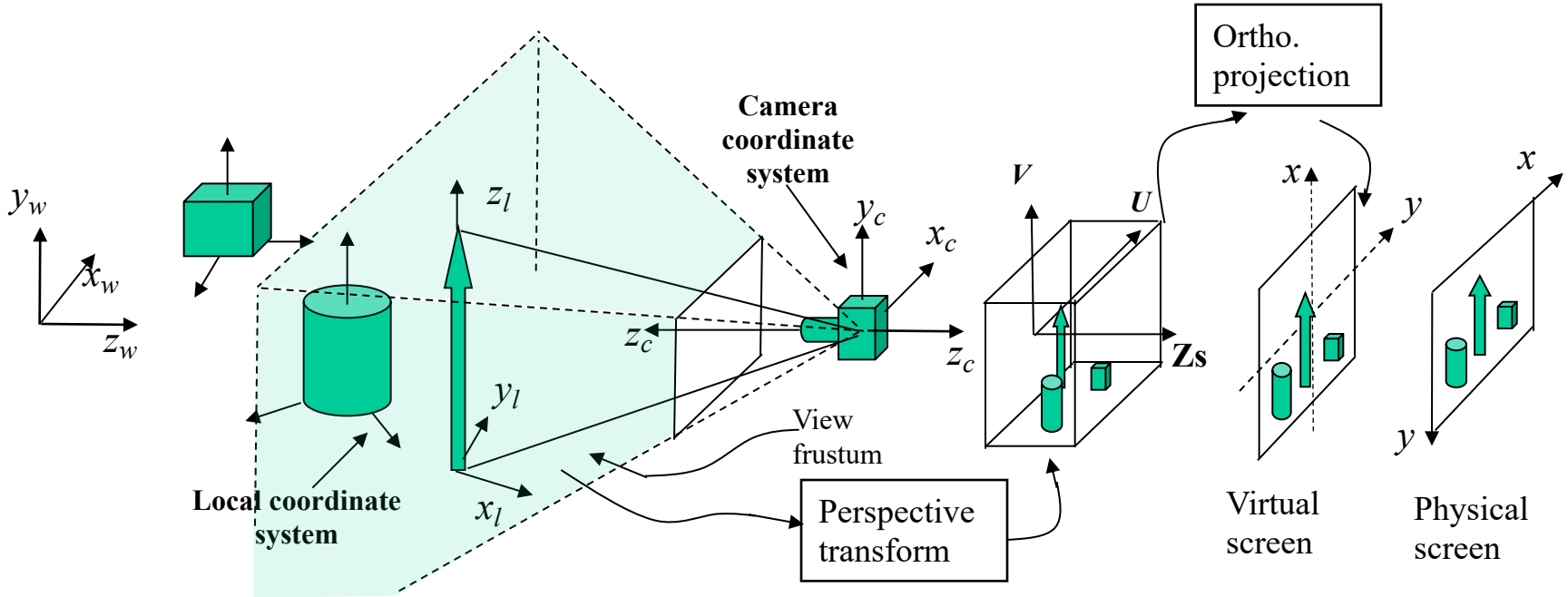
Lecture 01: 3D Graphics APIs and WebGL Pipeline

Overview

- An overview of 3D graphics pipeline
- Standards and APIs
- WebGL pipeline/processes

From Models to Graphics – Graphics Pipeline

- The conversion of 3D models (normally numeric) to graphics (renderings - images or videos) is carried out in a pipeline like process – polygon data is input at one end and graphics is output at the other.



Components of Pipeline

- 3D model: usually in form polygon meshes, e.g., triangle strips, of vertices.
- Coordinate systems: local, world, camera, screen coordinates systems
- Transformations: specify the spatial relationship between various coordinate systems and modify position & orientation of object with respect to those coordinate systems
- Various operations, clipping, back-surface culling, z-buffer algorithm, etc.

Transformations and the Pipeline

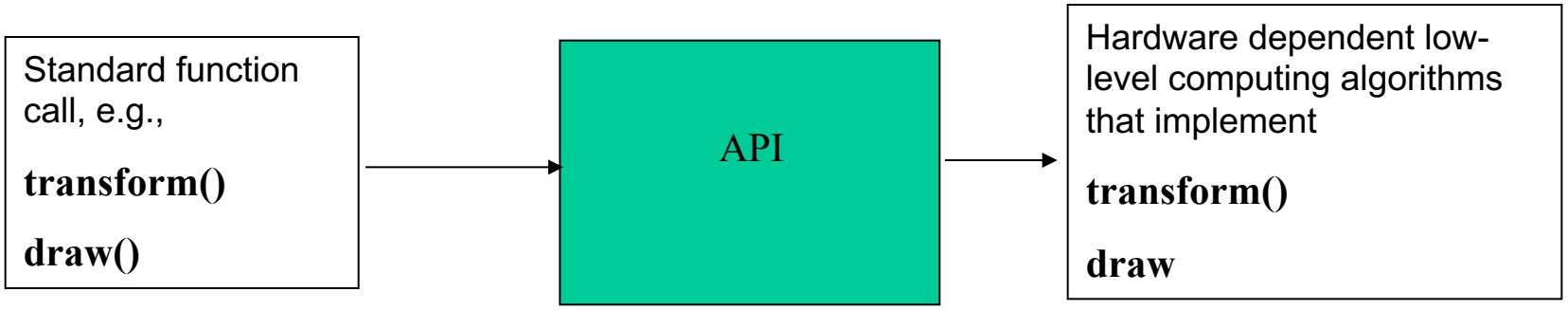
- The spatial relationships between the coordinate systems, i.e., the positions and poses, are defined by transformations (rotations and translations).
  - Defined as matrices.
  - Transformations mean matrix operations.
- The vertices of a model will go through the following transformations:
  - Local coordinates→world coordinates→camera coordinates (back surface culling. Polygon clipping can also be done at this stage) → 3d screen (z-buffer algorithm for hidden surface removal, polygon clipping) → virtual screen → physical display

Cont'd

- These processing steps are fixed.
- Very efficient algorithms for them are implemented in software and hardware as standard.
- They form the 3D graphics pipeline – the processing procedure and algorithms for rendering images.

3D Graphics APIs

- 3D graphics APIs (Application Programming Interface) specify 3D graphics operations, such as transformations, drawing, rendering etc.



Cont'd

- In 1980s, developing software that works on different graphics hardware had been a big problem.
- The aim of having standard APIs for graphics systems is to improve the software portability – make graphics implementations and applications hardware independent
  - Software developers need standardised ways to implement typical graphics operations/functions, e.g., transformation
  - Hardware manufacturers need a standard to develop hardware that support standard graphics functions

APIs – OpenGL

- GL – Graphics Library
  - Initially implemented as the graphics library (IRIS GL) on Silicon Graphics workstations (SGI) in 1991.
  - GL supported fast real-time rendering
  - GL was soon extended to other hardware systems (Open GL) and became the de facto standard.
  - OpenGL is the hardware-independent API that interfaces between application programs and hardware structure
  - Efficient processing of 3-dimensional applications.
  - Implemented for different high-level languages, e.g., C, C++, VB, Fortran

APIs - Direct3D

- It is a 3D graphics API for Windows platform and game consoles (XBox)
- It is one of the graphics APIs in DirectX (the other one is DirectDraw, for 2D graphics)
- Direct3D Supports various hardware acceleration
- A main competitor of OpenGL

APIs – WebGL

- WebGL is an API specifically developed for authoring 3D graphics for web applications in JavaScript.
- It is a variant of OpenGL:

  OpenGL → OpenGL ES → WebGL

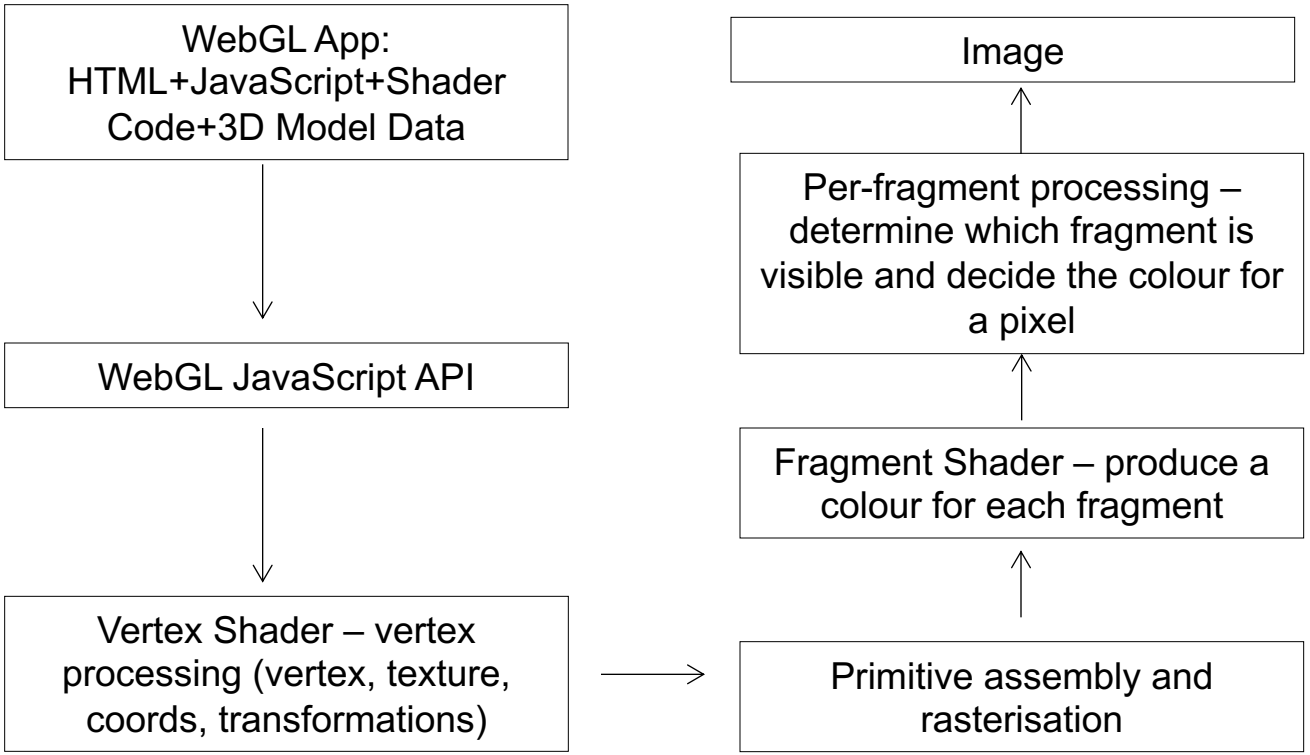
- Most browsers support it, e,g., Firefox, Chrome, Safari, IE/Edge, Opera

Advantages of WebGL

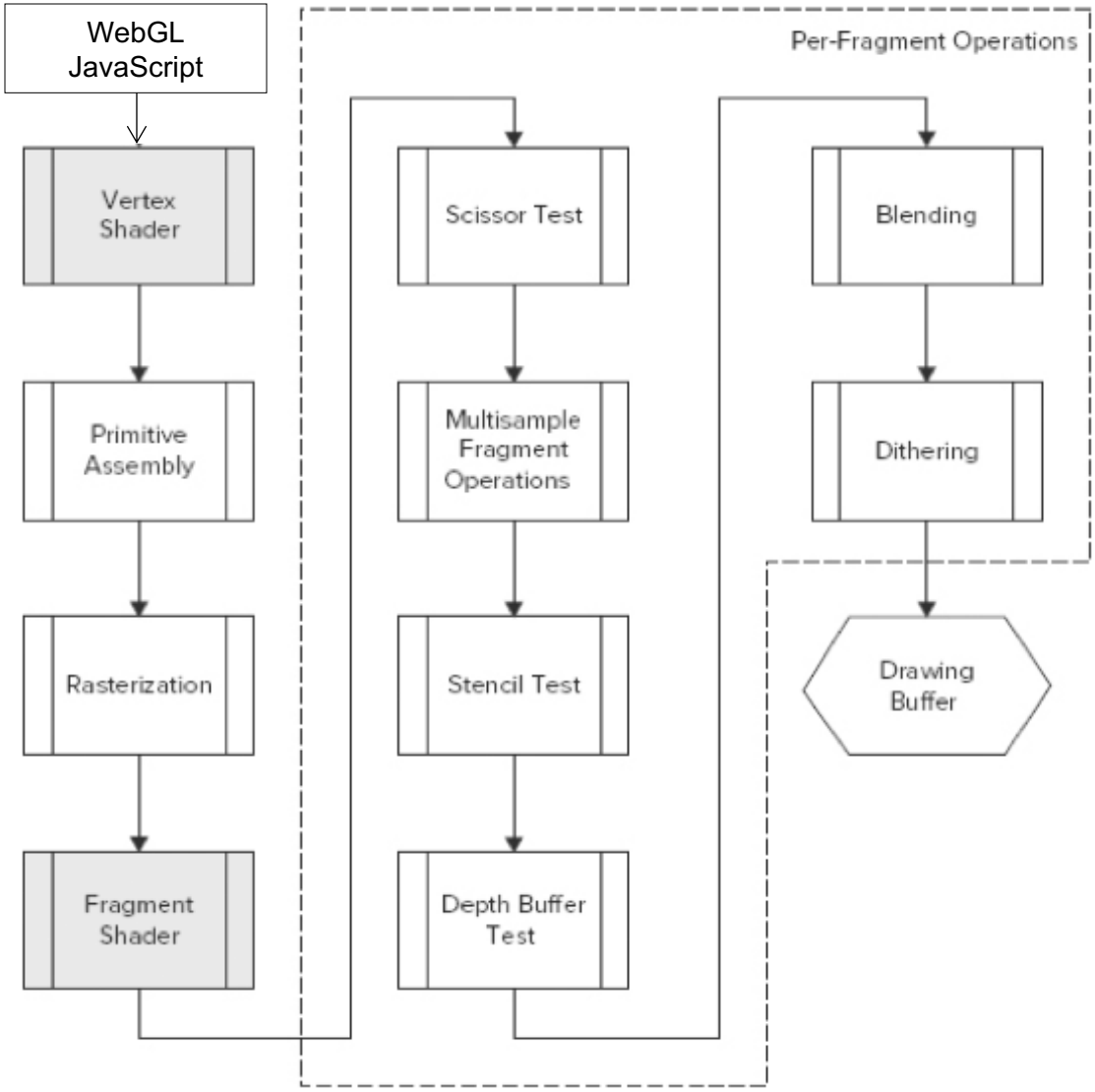
- Compared with native graphics applications, WebGL apps are cheap and easy to distribute to a lot of users.
- It is easier to have cross-platform support (i.e., to work on Windows, Mac OS, Linux, and so on) since WebGL runs natively in the browsers that support it; no plug-in is needed.
- It takes advantage of the graphics hardware to accelerate the rendering, which means it is really fast.
- It is an open standard that everyone can implement or use without paying royalties to anyone.

WebGL Pipeline

- WebGL follows idea of the general graphics pipeline and its implementation is similar to the OpenGL pipeline



Detailed Processes



Shaders

- In the graphics pipeline, to project the shapes with correct colours, you need to determine the effect of light on different materials – a process called shading.

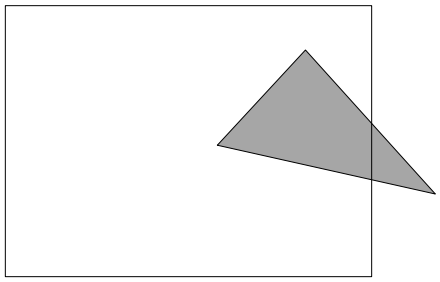
- For WebGL, the shading is done in two stages:
  - Vertex shader.
  - Fragment shader

Vertex Shader

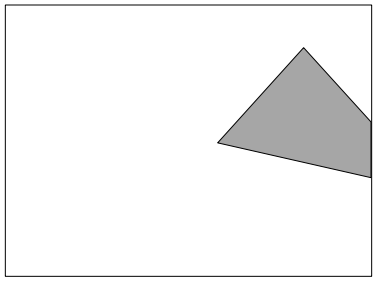
- The vertex shader is where the 3D modeling data (e.g., the vertices) first end up after they are sent in through the pipeline via JavaScript API.
- The vertex shader determines shade/colour **for a vertex**. Notice that it works on a single vertex at a time.
- Vertex shader is programmable - its source code (in OpenGL shading language) is written by the programmer and sent in through the JavaScript API.
- Vertex shader implements various transformations: it often transforms the vertex by multiplying it with a transformation matrix to place objects at a specific position in the scene.

Primitive Assembly

- After the vertex shader, the WebGL pipeline needs to assemble the shaded vertices into individual geometric primitives such as triangles, lines, or point sprites.
- Then the triangles, lines, or point sprites are clipped against the transformed view frustum.
  - Primitives outside the view frustum are completely removed, and primitives partly in the view frustum are clipped so the parts that are outside the view frustum are removed.
  - The items within the frustum are potentially visible on the screen for the moment and sent to the next step.



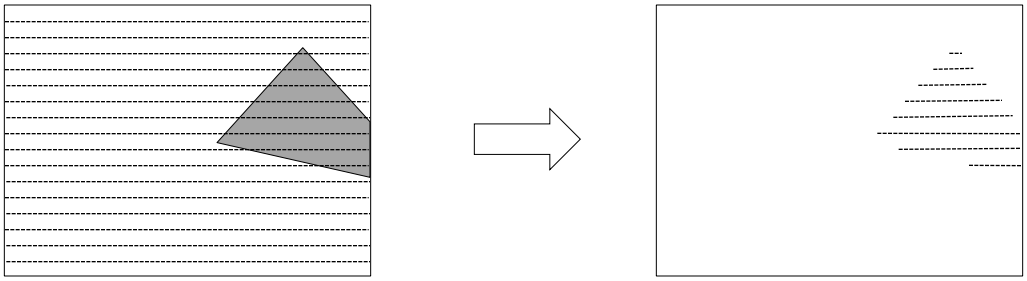
Assembled triangle



Clipped triangle

Rasterisation

- The next step in the pipeline is to convert the primitives inside the view frustum to fragments via rasterisation.



Rasterisation

Fragments

- A fragment is potentially a pixel that could finally be drawn on the screen. A fragment may and may not become an actual pixel on the screen.
- Fragments are sent to the fragment shader for further processing.

Fragment Shader

- Fragment shader is the second programmable stage of the pipeline where fragment-wise operations are done.
- Fragment shader assigns each fragment a shade that is calculated from vertex shades or texture via linear interpolation.
- Remember that not all fragments become pixels in the drawing buffer since some fragments might be discarded in the last steps of the pipeline. Fragments are called pixels when they are finally written to the drawing buffer.
- In other 3D rendering APIs, such as Direct3D from Microsoft, the fragment shader is called a pixel shader.

Per-Fragment Ops

- After the fragment shader, each fragment is sent to the next stage of the pipeline to be processed by the per-fragment operations.
  - Scissor Test**:** an additional level of clipping against a user defined rectangular region that further limits which fragments are writable to the drawing buffer. We normally do not need to do anything about it.
  - Stencil test: test a fragment against a stencil buffer (i.e., a mask) that contains per-pixel integer data which is used to add more control over (modify) which pixels are rendered. For example, to remove (a part of) an object from a scene, fill a stencil buffer with a cut out pattern (using zeros) for each pixel where the object is visible.

Cont'd

- Depth buffer test: The depth buffer test discards the incoming fragment depending on the value in the depth buffer (also called the Z-buffer).
  - The depth buffer stores the distance (z-value) from the viewer to the currently closest primitive.
  - For an incoming fragment, the z-value of that fragment is compared to the value of the depth buffer at the same position. If the z-value for the incoming fragment is smaller than the z-value in the depth buffer, then the new fragment is closer to the viewer than the pixel that was previously in the color buffer and the incoming fragment is passed through the test. If the z-value of the incoming fragment is larger than the value in the Z-buffer, then the new fragment is obscured by the pixel that is currently in the drawing buffer and therefore the incoming fragment is discarded.

Cont'd

- Blending: Blending combines the colour of the incoming fragment with the color of the fragment that is already available in the colour buffer at the corresponding position. Blending is needed for rendering transparent objects.
- Dithering: The last step before the drawing buffer is dithering. The colour buffer has a limited number of bits to represent each color. Dithering is used to arrange colors in such a way that an illusion is created of having more colors than are available. Dithering is most useful for a colour buffer that has few colors available.

Further Reading

- Chapter 1, Professional WebGL Programming: Developing 3D Graphics for the Web.