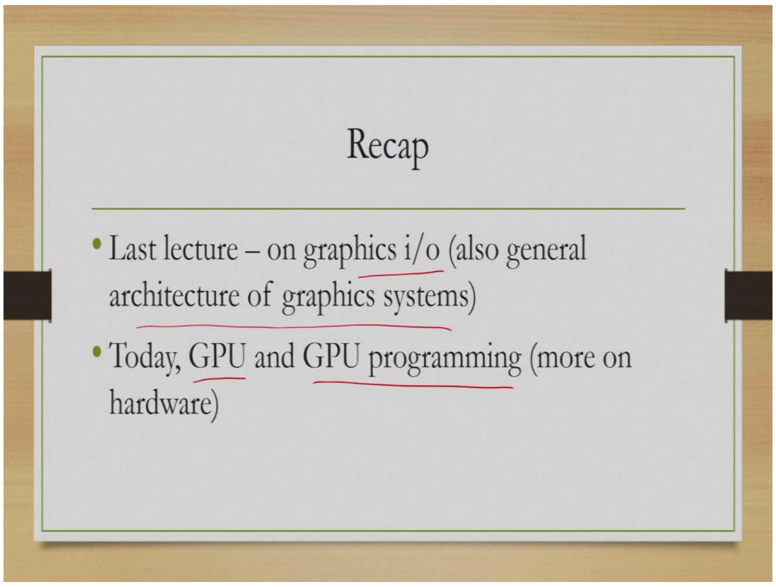
Computer Graphics
Professor Samit Bhattacharya
Computer Science and Engineering
Indian Institute of Technology, Guwahati
Lecture 30
Introduction to GPU and Shaders

Hello and welcome to lecture number 30 in the course, computer graphics. We have reached almost the end of the course.

(Refer Slide Time: 00:45)

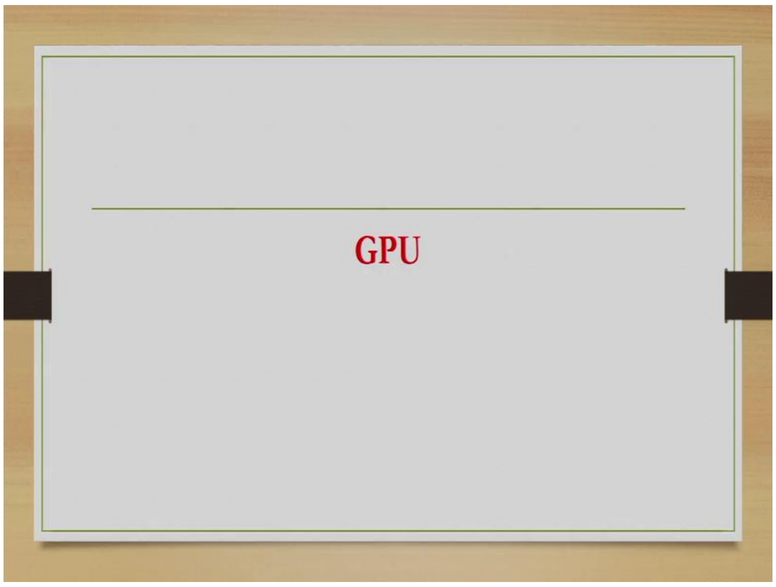


Recap

- Last lecture – on graphics i/o (also general architecture of graphics systems)
- Today, GPU and GPU programming (more on hardware)

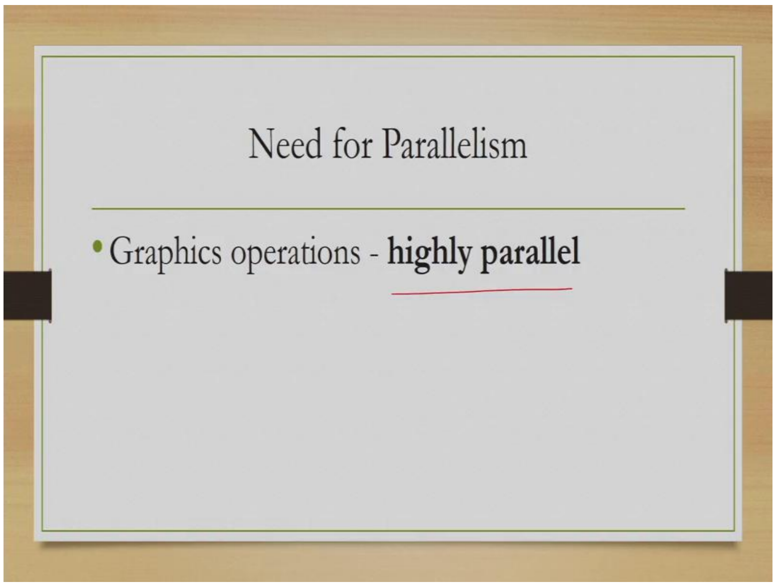
In the previous lecture we have learnt about basic graphics hardware. Also, we learnt about the graphics input and output devices and also general architecture of a graphic system. Now, we will continue our discussion on this graphics hardware and today we will cover basics of GPU and GPU programming, which are part of the graphics hardware.

(Refer Slide Time: 01:21)


GPU

So, let us start with GPU. What it is and how it is used to implement the pipeline.

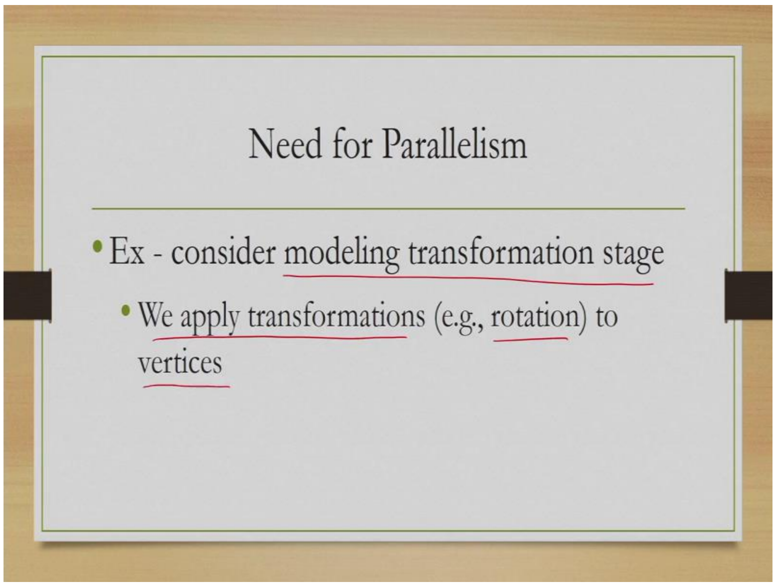
(Refer Slide Time: 01:40)


Need for Parallelism

- Graphics operations - **highly parallel**

One thing we should note in graphics is that the graphics operations are highly parallel in nature. That is a very crucial characteristics of graphics operations. Consequently, there is a need to go for parallel processing of these operations.

(Refer Slide Time: 01:56)



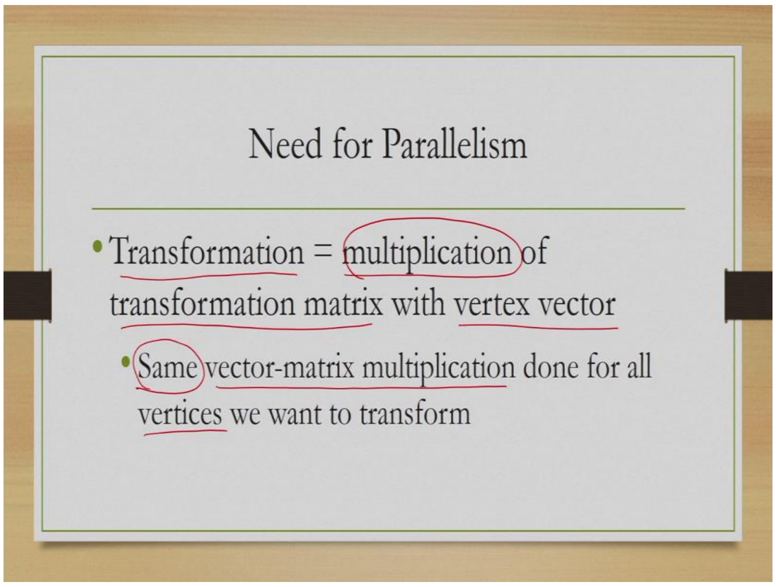
Need for Parallelism

- Ex - consider modeling transformation stage
  - We apply transformations (e.g., rotation) to vertices

For example, consider the modeling transformation stage. Remember that in this stage, what we do?

We convert or transform objects defined in their own or local coordinate system to a world coordinate scene. Now, how we do that, we apply transformations, for example rotations to the vertices that define the objects.

(Refer Slide Time: 02:34)



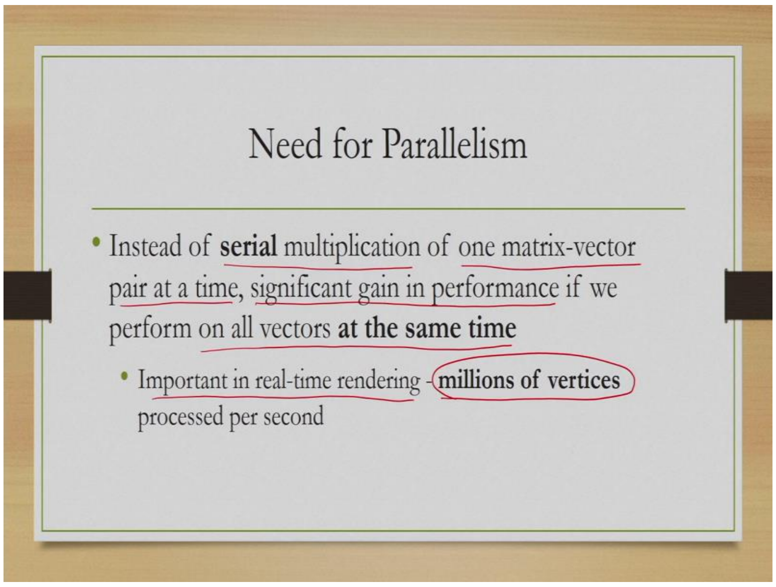
Need for Parallelism

- Transformation = multiplication of transformation matrix with vertex vector
  - Same vector-matrix multiplication done for all vertices we want to transform

And what is the transformation? If you may recollect from our earlier discussions, we define transformation as a multiplication of two things. One is a transformation matrix and the other is a vertex vector.

The thing to note here is that the same vector matrix multiplication is done for all vertices that we want to transform. That means we are essentially performing the same operation multiplication for all the vertices.

(Refer Slide Time: 03:26)



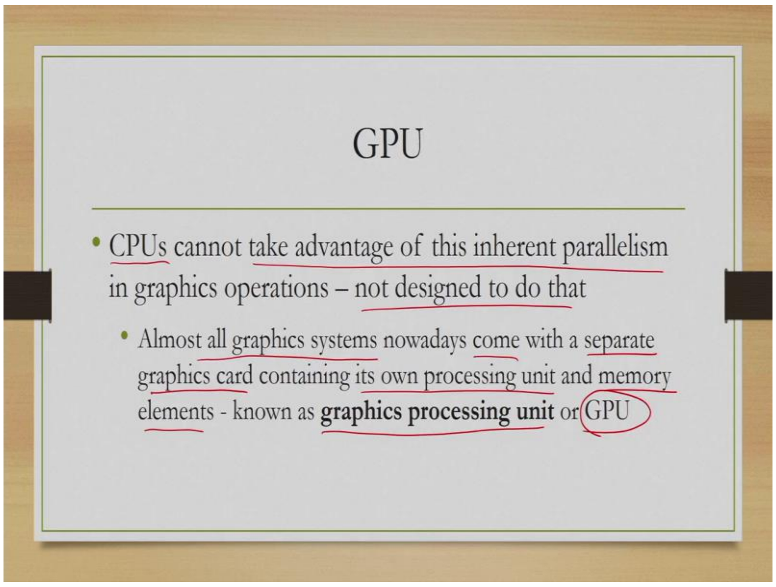
Need for Parallelism

- Instead of **serial** multiplication of one matrix-vector pair at a time, significant gain in performance if we perform on all vectors **at the same time**
  - Important in real-time rendering - **millions of vertices** processed per second

Now, we are given a set of vertices that define the objects. We can go for a serial multiplication where we perform one matrix vector multiplication at a time. However, that anyhow is not going to be very efficient.

Because essentially, we are performing the same operation, so instead of going for serial multiplication, if we can perform the same operation on all the vectors at the same time that is in parallel then we are going to have a significant gain in performance. And this is very important in real time rendering of scenes because typically we need to process millions of vertices at a time or millions of vertices per second. Therefore, if we can process all these millions of vertices parallelly then we are going to have huge gain in performance.

(Refer Slide Time: 04:40)



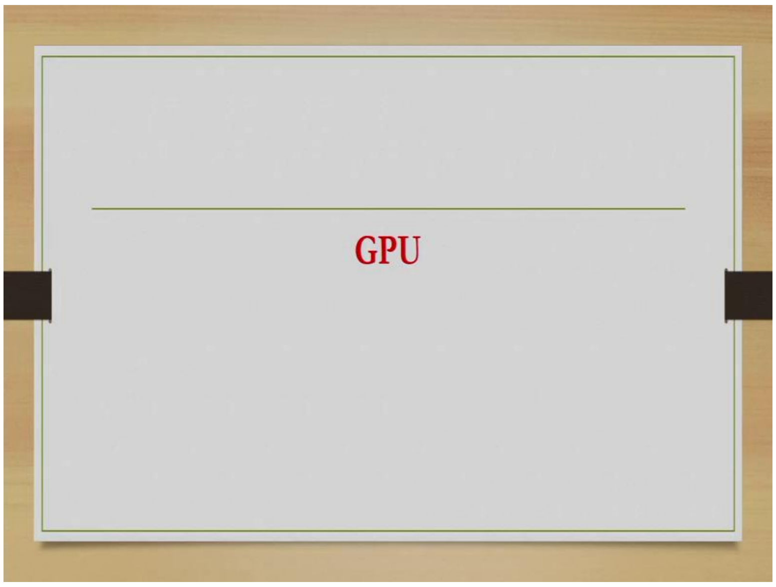
GPU

- CPUs cannot take advantage of this inherent parallelism in graphics operations – not designed to do that
  - Almost all graphics systems nowadays come with a separate graphics card containing its own processing unit and memory elements - known as **graphics processing unit** or GPU

If we are performing these operations using our CPU then we cannot take advantage of these inherent parallel nature of the graphics operations. Because CPUs are not designed for that. In order to address this issue where we want to take advantage of these inherent parallelism, there is special purpose hardware that comes with our systems.

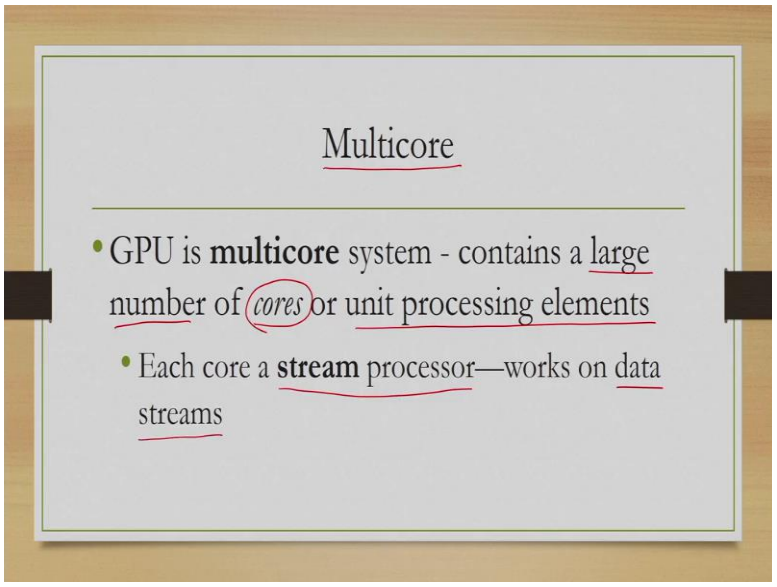
Almost all the graphics systems come with a separate graphics card, containing its own processing unit and memory elements. Now, this separate specialized hardware system is called graphics processing unit or GPU. So, essentially GPU means a specialized hardware which is used to perform graphic operation by exploiting the inherent parallelism that are there in graphics operations.

(Refer Slide Time: 06:08)



GPU

(Refer Slide Time: 06:24)

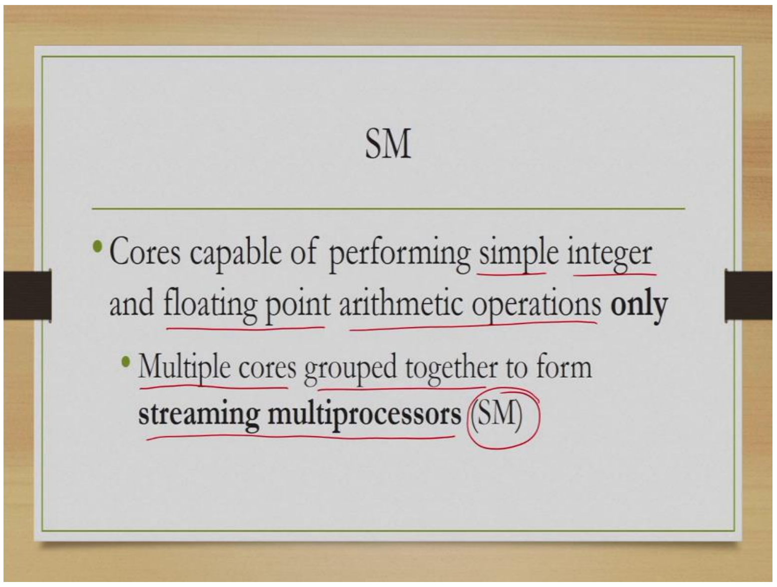


Multicore

- GPU is **multicore** system - contains a large number of cores or unit processing elements
  - Each core a **stream** processor—works on data streams

Let us go into the workings of GPU. Now, we have to note that GPU is a multicore system that means it contains a large number of cores or unit processing elements. Now, each of these cores or these unit processing elements is called a stream processor. Because it works on data streams, streams of input data.

(Refer Slide Time: 06:55)

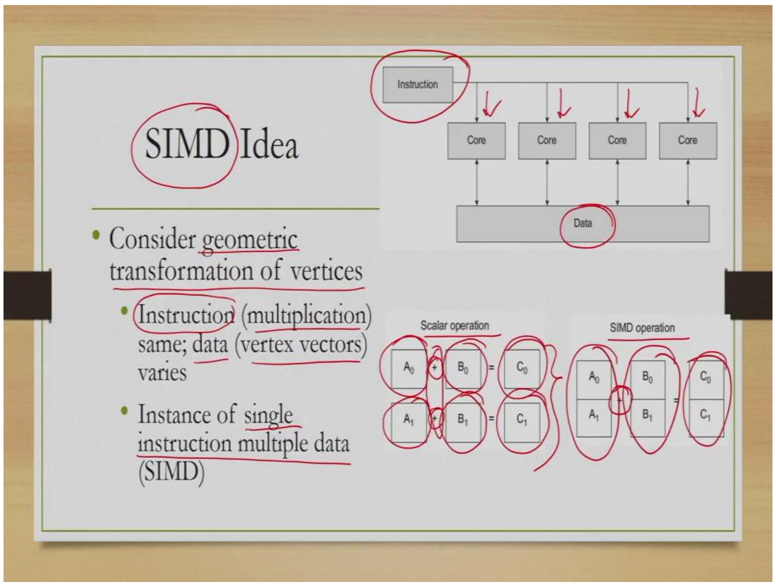


SM

- Cores capable of performing simple integer and floating point arithmetic operations **only**
  - Multiple cores grouped together to form **streaming multiprocessors** (SM)

Now, these cores are nothing but simple hardware capable of performing simple integer and floating-point arithmetic operations only. So, each core can perform arithmetic operations only, either integer arithmetic or floating point arithmetic. And multiple cores are grouped together to form another unit called streaming multiprocessors or SM. So, each core is called stream processor and many such cores are grouped together to form streaming multiprocessors.

(Refer Slide Time: 07:43)



SIMD Idea

- Consider geometric transformation of vertices
  - Instruction (multiplication) same; data (vertex vectors) varies
  - Instance of single instruction multiple data (SIMD)

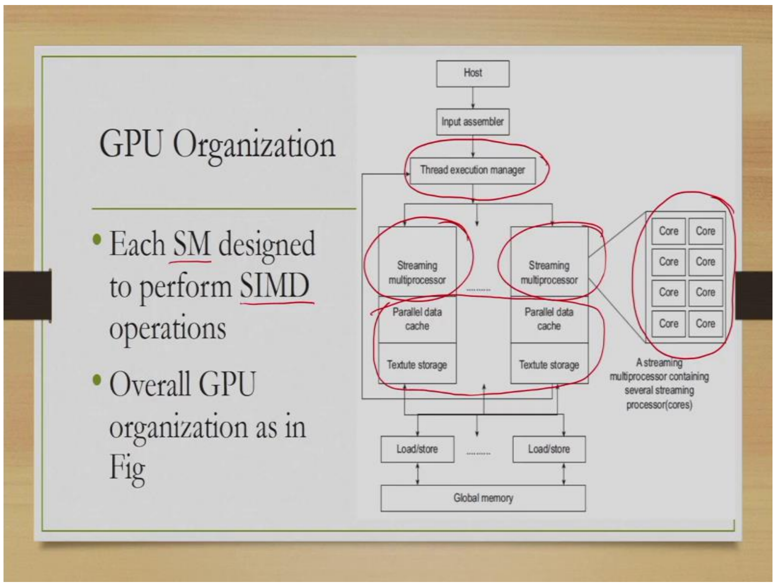
Instruction
Core
Core
Core
Core
Data
Scalar operation
A_0 + B_0 = C_0
A_1 + B_1 = C_1
SIMD operation
A_0 A_1 + B_0 B_1 = C_0 C_1

Now, this brings us to the idea of SIMD, note the term. To understand, let us consider one example, geometric transformation of vertices that we were discussing earlier. So, here our instruction is same which is multiplication. Now, the data on which this instruction operates varies, because the vertex vectors vary. Although the transformation matrix remains the same. So, then here what we are doing, we are having single instruction working on multiple data.

This is the idea of SIMD or Single Instruction Multiple Data, and the GPU streaming multiprocessors are essentially examples of SIMD. So, how it works, here as you can see, we have same instruction given to all the cores and the cores take data which may be different but the instruction will be same and the same instruction will operate on different data streams. Another illustration of this idea is given here, here if we do not consider SIMD then what happens. So we have two addition operations on two data streams.

A_0 B_0 and A_1 B_1, so there will be two separate instructions for performing these two separate additions giving the output of C_0 and C_1, this is normal operation. In case of SIMD what happens is that we have this as data streams and the instruction is single. Here note that we have two instructions working on two separate data streams. Here we have a single instruction working on both the data streams to give us the desired output. That is the idea of SIMD. So, you now know that GPUs contain SMs or streaming multiprocessors which work based on the idea of SIMD.

(Refer Slide Time: 10:25)

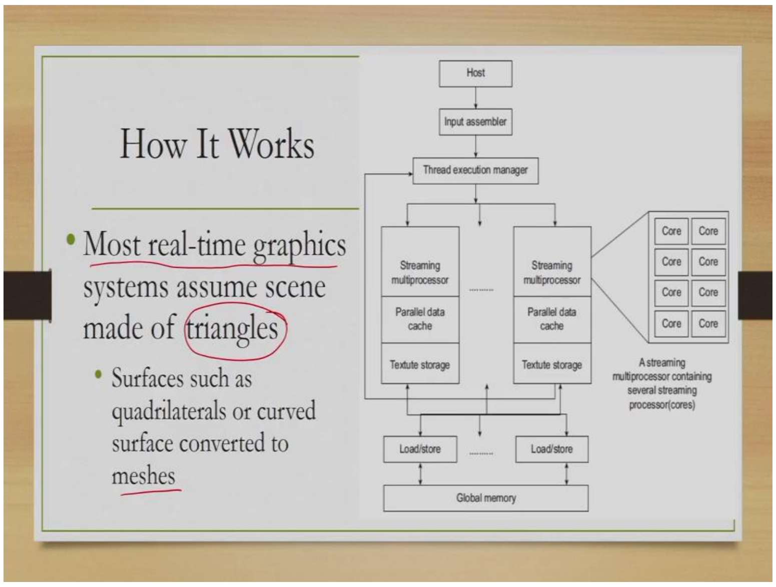

GPU Organization

- Each SM designed to perform SIMD operations
- Overall GPU organization as in Fig

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Textute storage
Streaming multiprocessor
Parallel data cache
Textute storage
Core
A streaming multiprocessor containing several streaming processor(cores)
Load/store
Load/store
Global memory

Then let us have a look at how the GPUs are organised. As I said each streaming multiprocessor is designed to perform SIMD operations. So and we have many such streaming multiprocessors, as shown here. Then we have some specialized memory, purpose of which will be explained to you shortly, and other components to manage this parallel processing. Each streaming multiprocessor contains multiple streaming processors or course as shown here. And each core is capable of performing simple integer or floating-point arithmetic operations only.

(Refer Slide Time: 11:33)

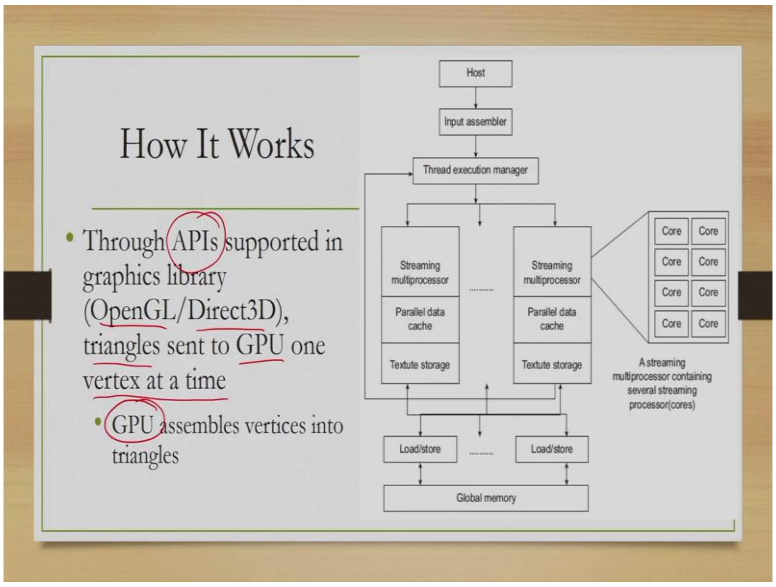

How It Works

- Most real-time graphics systems assume scene made of triangles
  - Surfaces such as quadrilaterals or curved surface converted to meshes

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Textute storage
Streaming multiprocessor
Parallel data cache
Textute storage
Core
A streaming multiprocessor containing several streaming processor(cores)
Load/store
Load/store
Global memory

So, that is broadly what are there in GPU, streaming multiprocessors and dedicated memory units plus additional components to manage this parallel processing. Now, let us try to understand how the graphics operations work in the GPUs. First thing we should note is that most real time graphic systems assume that scene is made of triangles. So, we actually convert any surface into triangles or triangular meshes. This point we have already discussed earlier when we were talking about object representation.

(Refer Slide Time: 12:34)



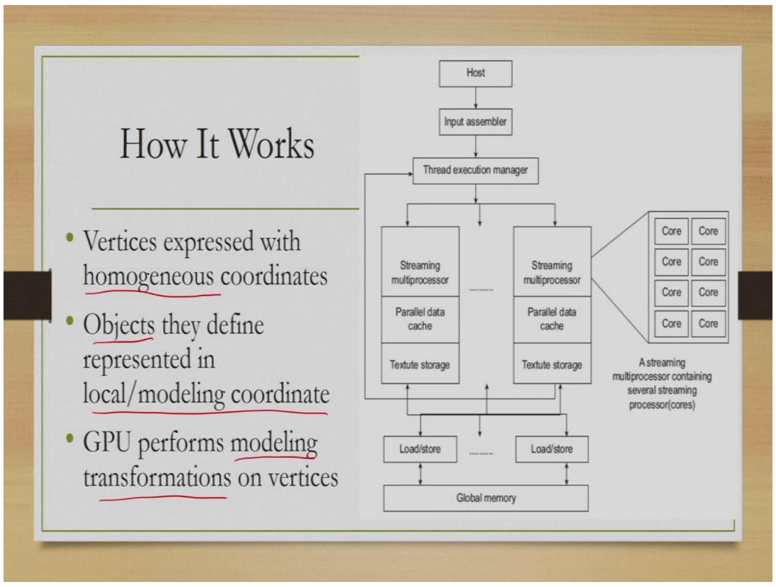
How It Works

- Through APIs supported in graphics library (OpenGL/Direct3D), triangles sent to GPU one vertex at a time
    - GPU assembles vertices into triangles

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Textute storage
Streaming multiprocessor
Parallel data cache
Textute storage
Core
A streaming multiprocessor containing several streaming processor(cores)
Load/store
Load/store
Global memory

Now, given that triangular mesh information what happens is that, those dedicated APIs which are provided in the graphics library such as OpenGL or Direct3D, these triangles are send to GPU, one vertex at a time serially and GPU assembles them into triangles.

(Refer Slide Time: 13:03)

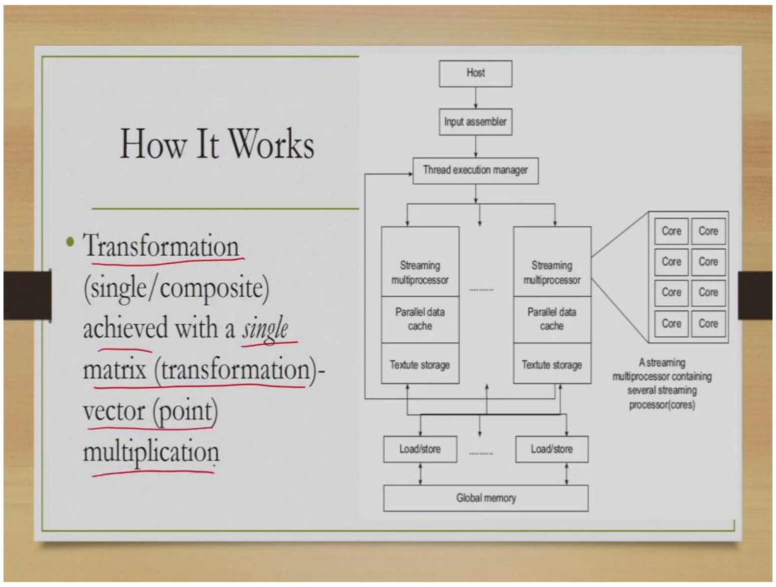


How It Works

- Vertices expressed with homogeneous coordinates
- Objects they define represented in local/modeling coordinate
- GPU performs modeling transformations on vertices

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Textute storage
Streaming multiprocessor
Parallel data cache
Textute storage
Core
Load/store
Load/store
Global memory
A streaming multiprocessor containing several streaming processor(cores)

Also, we should note here that the vertices are represented with homogeneous coordinate system. So the vertices are represented in the homogeneous coordinate system.



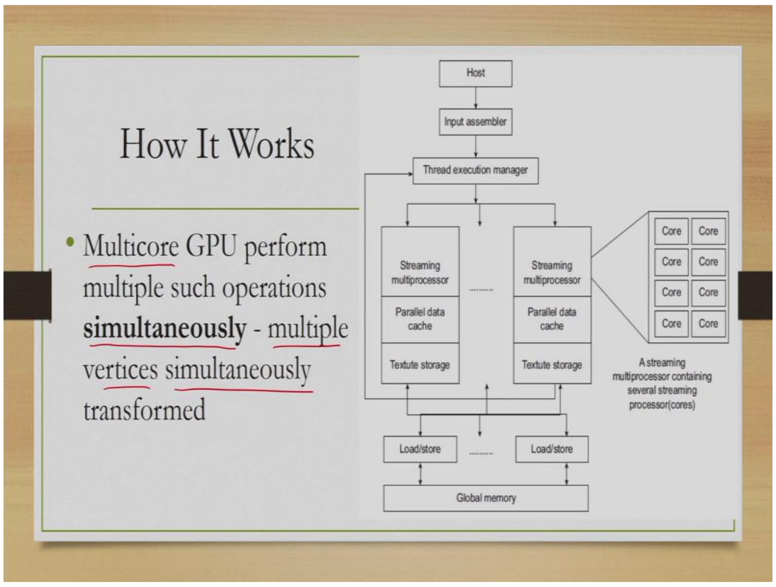
How It Works

- Transformation (single/composite) achieved with a *single* matrix (transformation)-vector (point) multiplication

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Textute storage
Streaming multiprocessor
Parallel data cache
Textute storage
Core
Load/store
Load/store
Global memory
A streaming multiprocessor containing several streaming processor(cores)

And so we are dealing here with the very first stage that is object definition, so these objects are defined in their local or modeling coordinate systems. Then the GPU performs all the stages, so first it performs modeling transformation on vertices that is the first stage of processing.

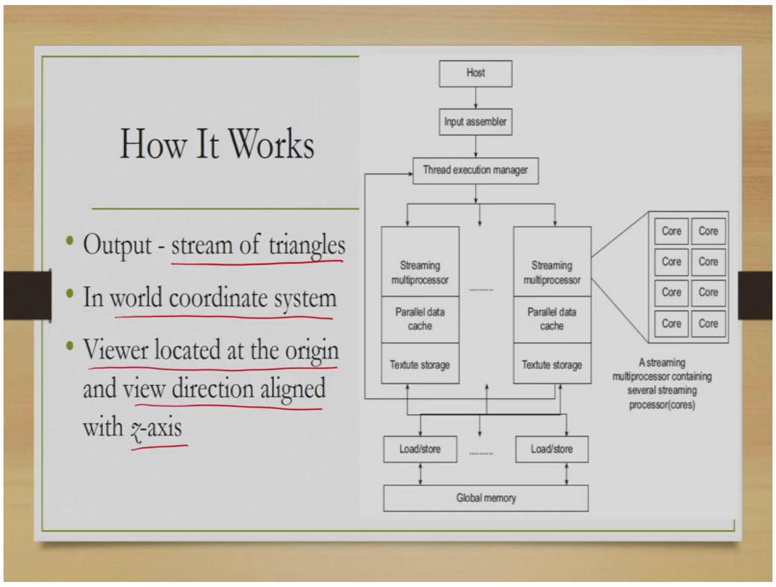
(Refer Slide Time: 13:54)


How It Works

- Multicore GPU perform multiple such operations **simultaneously** - multiple vertices simultaneously transformed

And as we have explained earlier this transformation is achieved with a single transformation matrix and vector point multiplication operation.

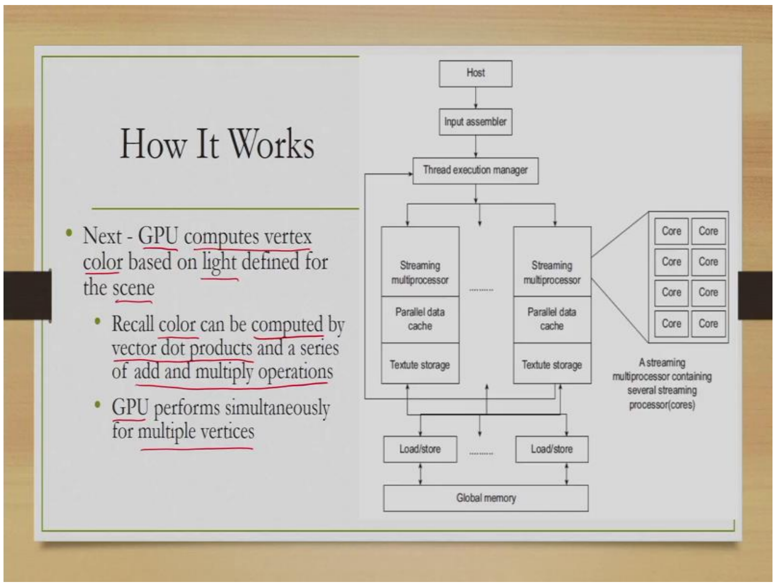
(Refer Slide Time: 14:15)


How It Works

- Output - stream of triangles
- In world coordinate system
- Viewer located at the origin and view direction aligned with z-axis

As we have noted earlier, the multicore GPU performs such operations simultaneously or parallely, so essentially multiple vertices are transformed simultaneously at the same time. Is not that one after another we are performing the multiplications. So, what we get after multiplication that is stream of triangles but this time, they are defined in world coordinate system, which is the purpose of modelling transformation stage. It is also assumed that the viewer is located at the origin of the world coordinate system and view direction is aligned with the z axis. This is the assumption using which the hardware is designed.

(Refer Slide Time: 15:17)



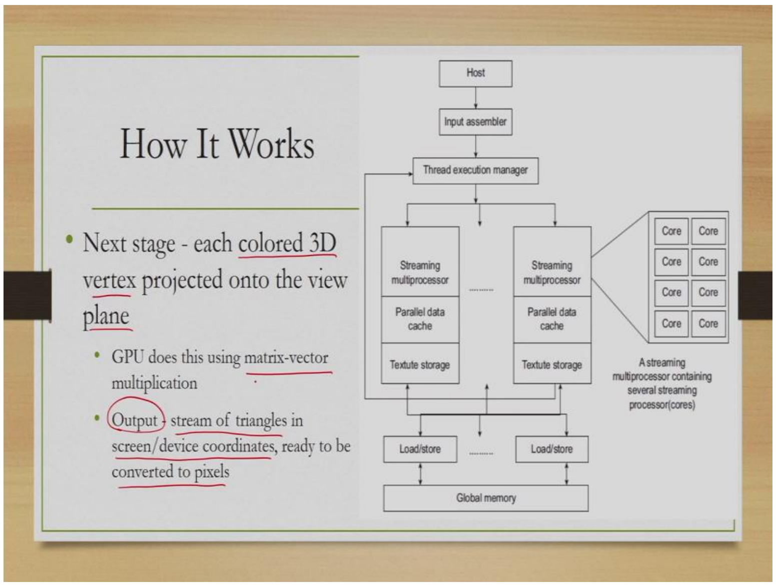
How It Works

- Next - GPU computes vertex color based on light defined for the scene
  - Recall color can be computed by vector dot products and a series of add and multiply operations
  - GPU performs simultaneously for multiple vertices

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Texture storage
Core
A streaming multiprocessor containing several streaming processor(cores)
Load/store
Global memory

So, after modeling transformation, GPU computes vertex colour or the lighting stage is performed. Now, this is done based on the light that is defined for the scene, so some light source is assumed and based on that light source this colouring is done. Now, why GPU is suitable for computing colours because if you may recollect our discussion on lighting, we have noted that colouring can be computed by vector dot products and a series of addition and multiplication operations. And these operations are performed simultaneously for multiple vertices by the GPU because it is designed in that way, so again here we are exploiting the inherent nature of graphics operations that is parallelism.

(Refer Slide Time: 16:38)



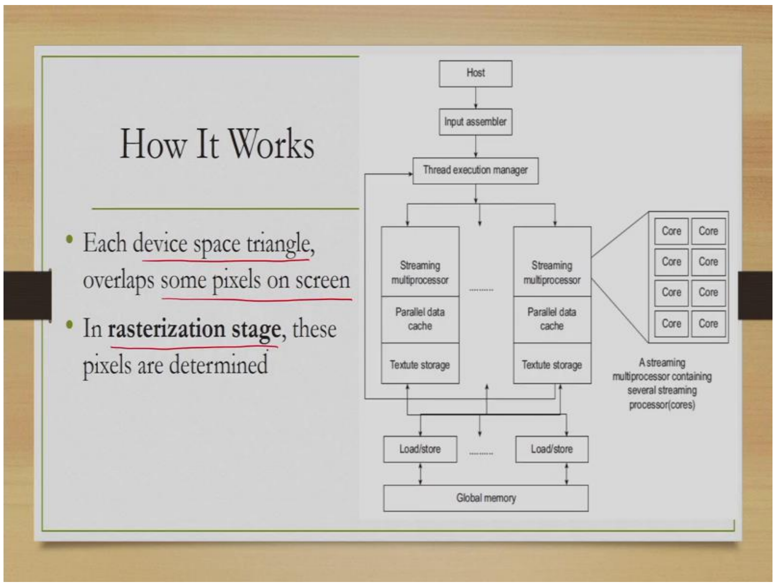
How It Works

- Next stage - each colored 3D vertex projected onto the view plane
  - GPU does this using matrix-vector multiplication
  - Output stream of triangles in screen/device coordinates, ready to be converted to pixels

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Texture storage
Core
A streaming multiprocessor containing several streaming processor(cores)
Load/store
Global memory

After colouring, each coloured 3D vertex is projected on to the blue plane. And that again is done using matrix vector multiplication, we have already noted this before during our discussion on projection transformation and the output that we get is stream of triangles in the screen or device coordinates ready to be converted to pixels. Now, note here that this projection actually involves view transformation also. Which we have not explicitly mentioned here as well as window to view put transformation. All these transformations we can club together by multiplying the corresponding transformation matrices to get a single transformation matrix.

(Refer Slide Time: 17:46)

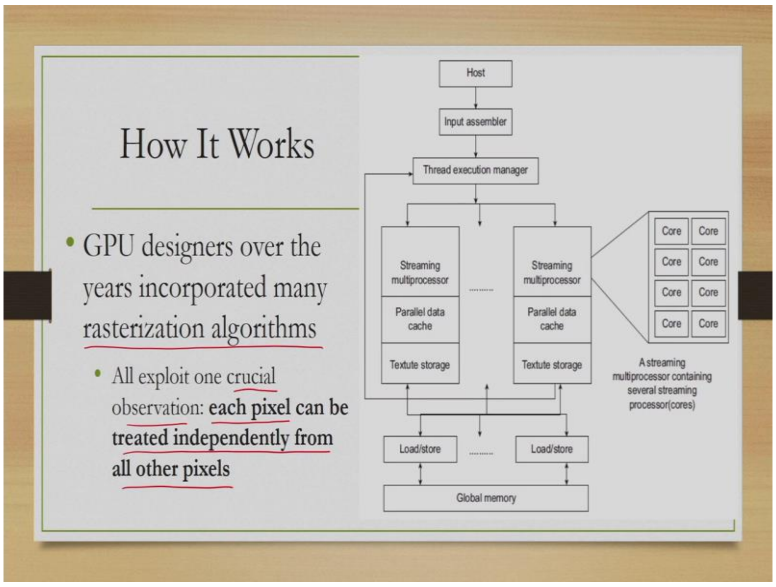


How It Works

- Each device space triangle, overlaps some pixels on screen
- In **rasterization stage**, these pixels are determined

So, after that stage we get the device space triangles and we now go for rasterization or scan conversion. So here it may be noted that each device space triangle overlaps some pixels on the screen that means those pixels are part of the triangles. In the rasterization stage these pixels are determined.

(Refer Slide Time: 18:20)



How It Works

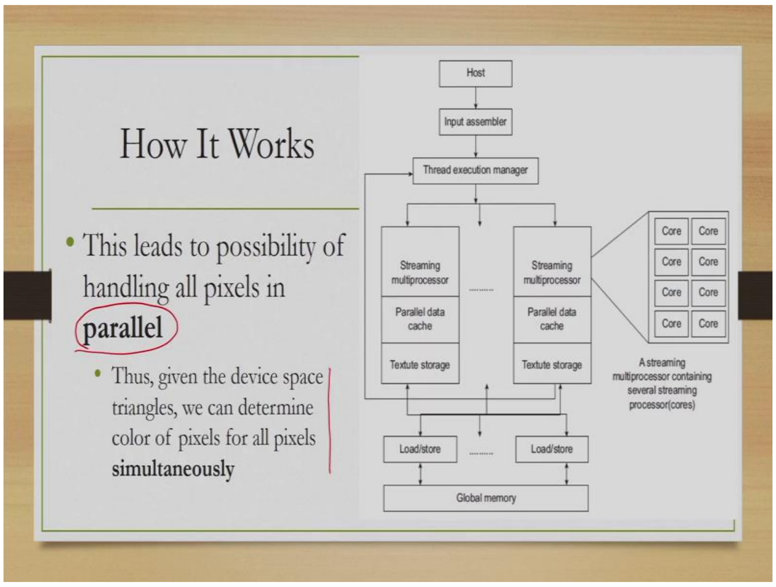
- GPU designers over the years incorporated many rasterization algorithms
  - All exploit one crucial observation: **each pixel can be treated independently from all other pixels**

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Texture storage
Load/store
Global memory
Core
A streaming multiprocessor containing several streaming processor(cores)

Now, the GPU designers who developed GPUs over the years incorporated many such rasterization algorithms, we have already discussed few in our discussions on rasterization.

Now, these algorithms exploit one crucial observation that is each pixel can be treated independently from all other pixels. So, it is not necessary to treat the pixels as dependent on each other they can be treated independently.

(Refer Slide Time: 19:01)



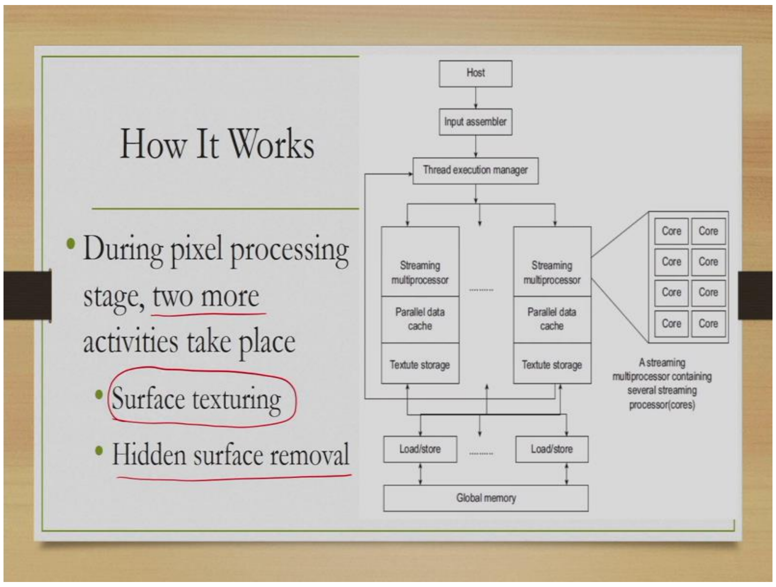
How It Works

- This leads to possibility of handling all pixels in parallel
  - Thus, given the device space triangles, we can determine color of pixels for all pixels **simultaneously**

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Texture storage
Load/store
Global memory
Core
A streaming multiprocessor containing several streaming processor(cores)

Accordingly, the pixels can be rasterised parallely, so you can use this inherent parallelism to rasterize all the pixels simultaneously. And that is one big advantage of having GPU, we do not have to process one pixel at a time instead we can process all the pixels together to get quick result, quick output.

(Refer Slide Time: 19:36)



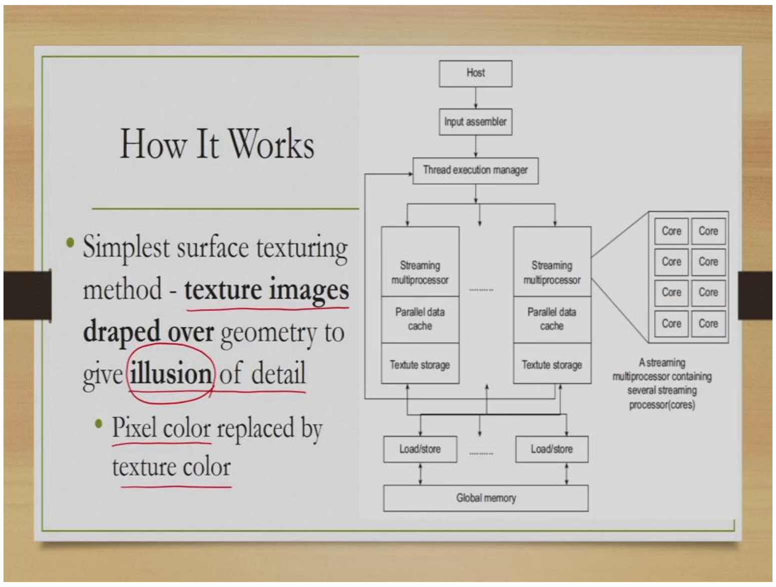
How It Works

- During pixel processing stage, two more activities take place
  - Surface texturing
  - Hidden surface removal

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Textute storage
Streaming multiprocessor
Parallel data cache
Textute storage
Core
Load/store
Load/store
Global memory
A streaming multiprocessor containing several streaming processor(cores)

Now, if you may recollect the 3D graphics pipeline, during pixel processing stage there are two more stages that are there, two more activities that are there, one is surface texturing or assigning patterns to the surface colours and second is hidden surface removal or HSR.

(Refer Slide Time: 20:03)



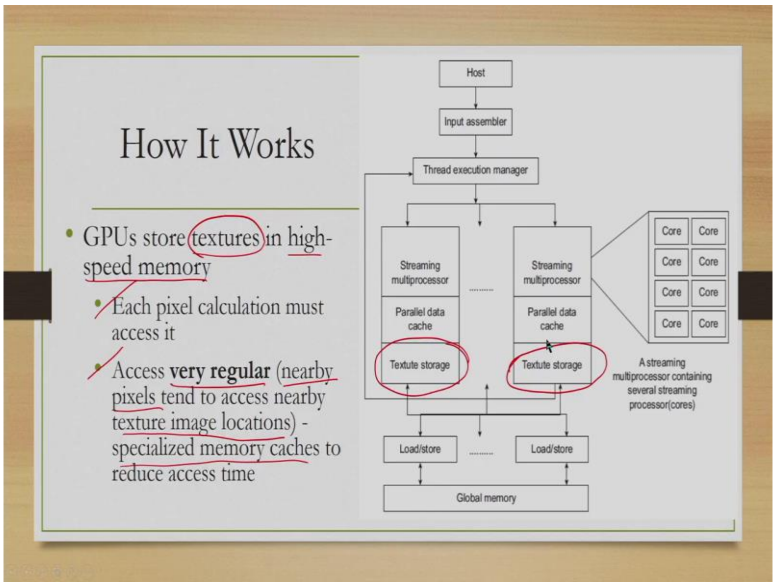
How It Works

- Simplest surface texturing method - **texture images draped over** geometry to give **illusion** of detail
  - Pixel color replaced by texture color

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Textute storage
Streaming multiprocessor
Parallel data cache
Textute storage
Core
Load/store
Load/store
Global memory
A streaming multiprocessor containing several streaming processor(cores)

Now, surface texturing idea is very simple here texture image is there which is actually imposed on the surface to give us the illusion of details. Note that it is only an illusion creation rather than actually computing a texture pattern, a simply replacing pixel colours with texture colour. That is the simplest idea we discussed earlier.

(Refer Slide Time: 20:40)

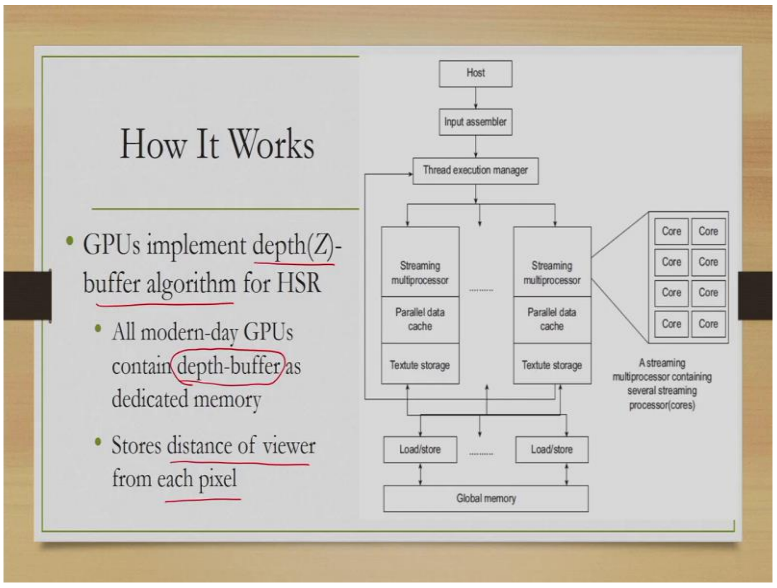


How It Works

- GPUs store textures in high-speed memory
  - Each pixel calculation must access it
  - Access **very regular** (nearby pixels tend to access nearby texture image locations) - specialized memory caches to reduce access time

Now, in order to that we need to store this texture images or texture maps. And since we need to access it, that texture images frequently, ideally they should be stored in high speed memory, so that access time is less. Now, this is because as we said earlier pixel calculations are very frequent and each pixel calculation must access this texture images. Secondly the access is usually very regular in nature. That means nearby pixels tend to access nearby texture images or texture image locations. So, to reduce the access time specialized memory cache is used to store the texture images as shown here in this figure. These are specialized memory locations in the GPU to store texture images.

(Refer Slide Time: 22:05)



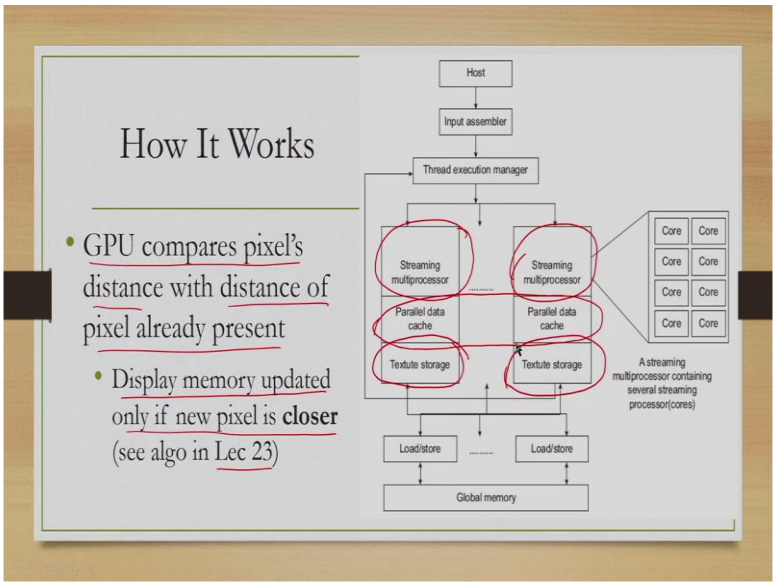
How It Works

- GPUs implement depth(Z)-buffer algorithm for HSR
  - All modern-day GPUs contain depth-buffer as dedicated memory
  - Stores distance of viewer from each pixel

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Texture storage
Core
A streaming multiprocessor containing several streaming processor(cores)
Load/store
Global memory

Also, we discussed earlier during our discussion on hidden surface removal, the idea of Z buffer algorithm or depth buffer algorithm. Now, that is implemented in GPUs and for that also typically GPUs are equipped with specialized memory element or depth buffers. And it stores distance of viewers from each pixel. So, that is typically part of the GPU.

(Refer Slide Time: 22:48)



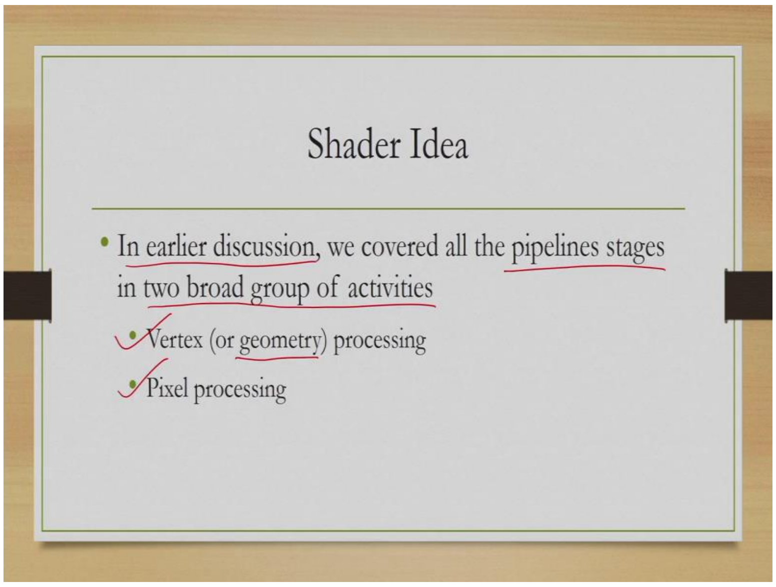
How It Works

- GPU compares pixel's distance with distance of pixel already present
  - Display memory updated only if new pixel is **closer** (see algo in Lec 23)

Host
Input assembler
Thread execution manager
Streaming multiprocessor
Parallel data cache
Texture storage
Core
A streaming multiprocessor containing several streaming processor(cores)
Load/store
Global memory

Now if you may recollect how the Z buffer works, so here also GPU compares pixels distance with distance of pixel already present that is it simply executes the algorithm and the display memory is updated only if the new pixel is closer. So, it implements the Z buffer algorithm, for more details you may refer to lecture 23. So, you have the streaming multiprocessors, each containing course, then these various data items for performing these simultaneous operations plus specialized texture storage that form GPU.

(Refer Slide Time: 24:07)



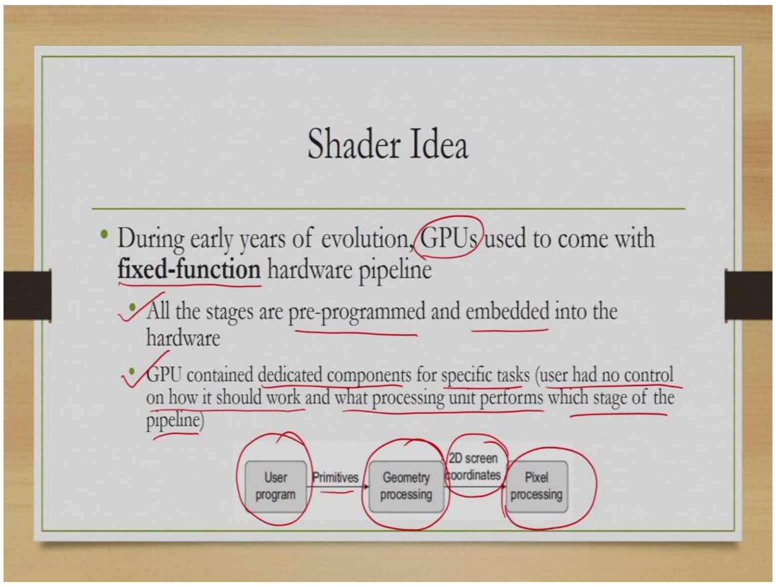
Shader Idea

- In earlier discussion, we covered all the pipelines stages in two broad group of activities
  - Vertex (or geometry) processing
  - Pixel processing

Now, there is one more concept that is the idea of shaders and shader programming. Let us try to understand this programming concept in brief at a very introductory level. In our earlier discussion on GPU what we discussed is that, how the GPUs implement pipeline stages. Now, in that discussion if you may have noted there are two broad group of activities, one is processing of vertices or vertex processing also called geometry processing.

Other one is processing of pixels. So, these two broad group of activities were discussed to explain the working of GPU.

(Refer Slide Time: 24:54)



Shader Idea

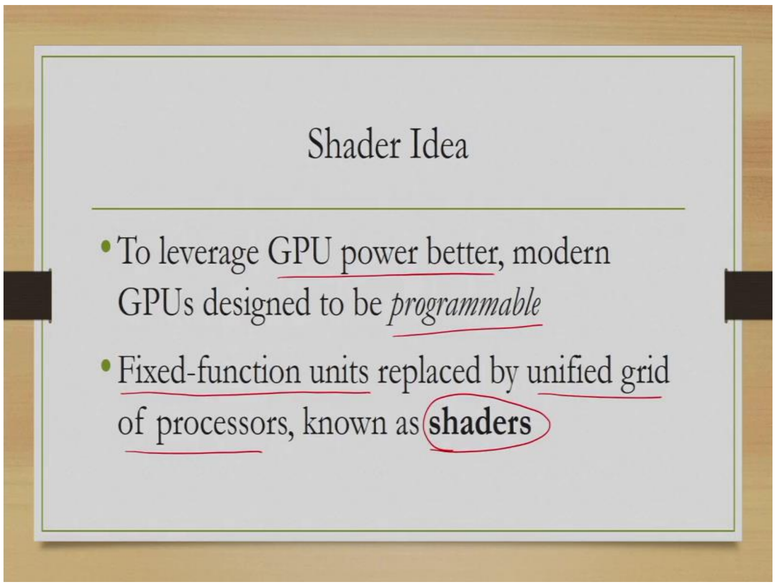
- During early years of evolution, GPUs used to come with **fixed-function** hardware pipeline
  - All the stages are pre-programmed and embedded into the hardware
  - GPU contained dedicated components for specific tasks (user had no control on how it should work and what processing unit performs which stage of the pipeline)

User program
Primitives
Geometry processing
2D screen coordinates
Pixel processing

Now, during the early years of GPUs they used to come with fixed function hardware pipeline, that means all the pipeline stages or all the stages that implement the pipeline are pre-programmed and embedded into the hardware. GPU content dedicated components for specific tasks and the user had no control on how this task should be performed and what processing unit performs which stage of the pipeline.

So, earlier GPU is used to come with this fixed function hardware that means everything was predetermined, which component of the GPU will deal with which part of the pipeline and the user had no control on it. So, the flow was typically like this from user program the primitives were sent, then components were there for geometry processing, output is 2D screen coordinates from there pixel processing starts and components were again fixed.

(Refer Slide Time: 26:11)

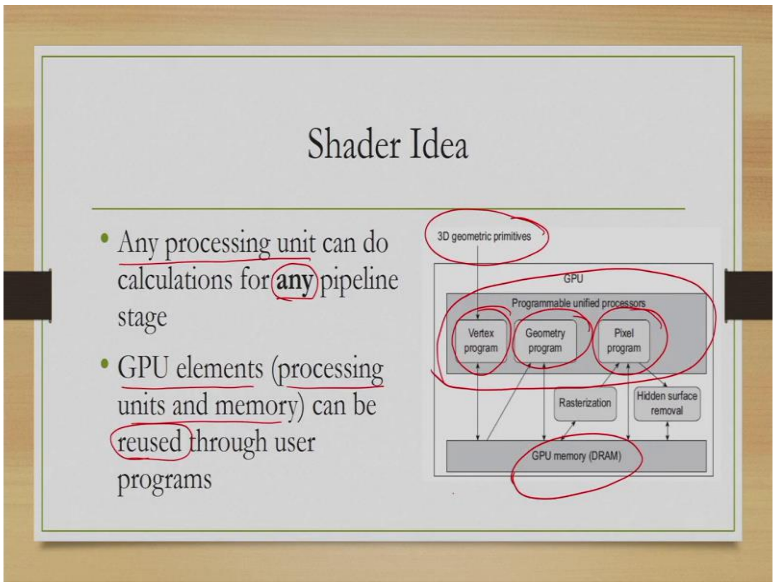


Shader Idea

- To leverage GPU power better, modern GPUs designed to be *programmable*
- Fixed-function units replaced by unified grid of processors, known as **shaders**

But then people realize that, that is actually reducing flexibility and because of that power of GPU was not fully utilized. To leverage the GPU power better, modern GPUs are designed to be programmable that means we can program them. Fixed function units are replaced by unified grid of processors known as shaders. So, earlier there were fixed function units, now there are unified grid of processors which are called shaders.

(Refer Slide Time: 27:05)



Shader Idea

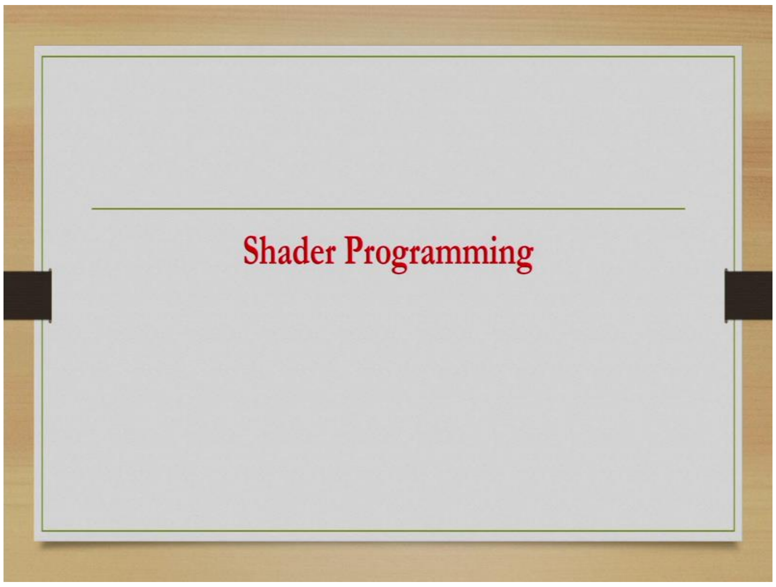
- Any processing unit can do calculations for **any** pipeline stage
- GPU elements (processing units and memory) can be reused through user programs

And any processing unit can be used for performing any pipeline stage calculation. And the GPU elements, that is the processing units and memory can be reused through user programs.

So, earlier we had fixed units for performing different stages, now we have common facilities which are reused to perform different stages and that is determined through programming.

Which portion and how the GPU elements namely the processing units and memory are used for performing operations related to a particular stage. The idea is shown here as you can see once the primitives are sent to the GPU, GPU as a common element, now subset of these common elements are used for different purposes as you can see here also the memory is also shared and reused.

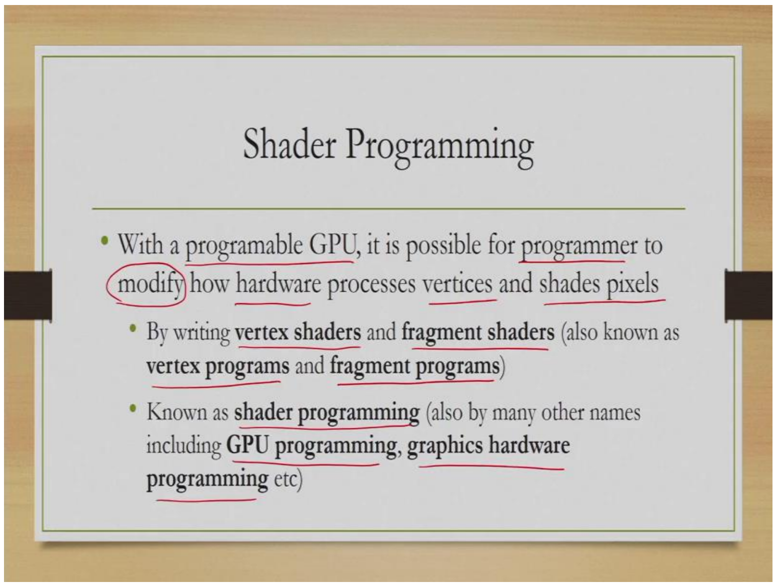
(Refer Slide Time: 28:35)



Shader Programming

Now, the idea is that we write programs to use GPU elements, these programs are called shader programs and the corresponding approach is called shader programming. Let us briefly go through the basics of shader programming.

(Refer Slide Time: 29:01)



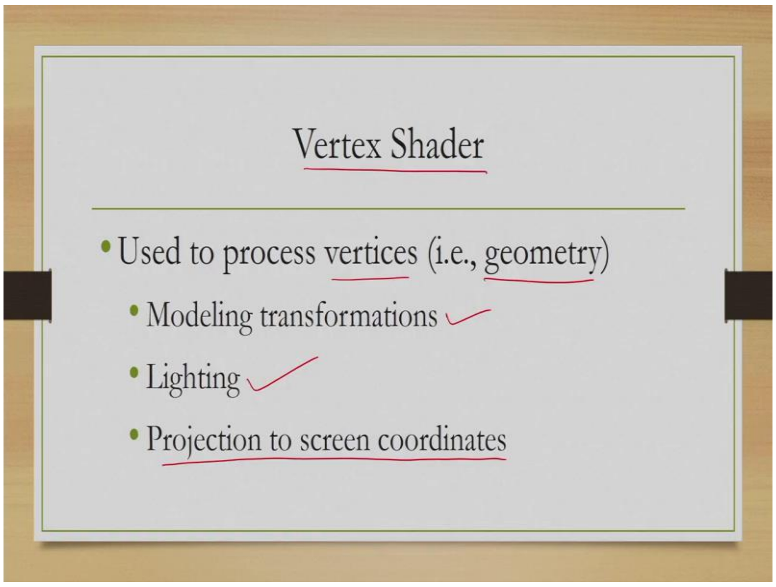
Shader Programming

- With a programable GPU, it is possible for programmer to modify how hardware processes vertices and shades pixels
  - By writing **vertex shaders** and **fragment shaders** (also known as **vertex programs** and **fragment programs**)
  - Known as **shader programming** (also by many other names including **GPU programming**, **graphics hardware programming** etc)

Now, with the programable GPU that we have just introduced it is possible for programmer to modify how the GPU hardware processes vertices and shades pixels, shades means assigns colour to the pixels.

This is possible by writing vertex shaders and fragment shaders, these are also called vertex programs and fragment programs. These are terms that you probably have come across with these are used to specify to the GPU how to use its hardware for specific purpose. And this approach as I said is known as shader programming and it has other names also such as GPU programming, graphics hardware programming and so on.

(Refer Slide Time: 30:12)

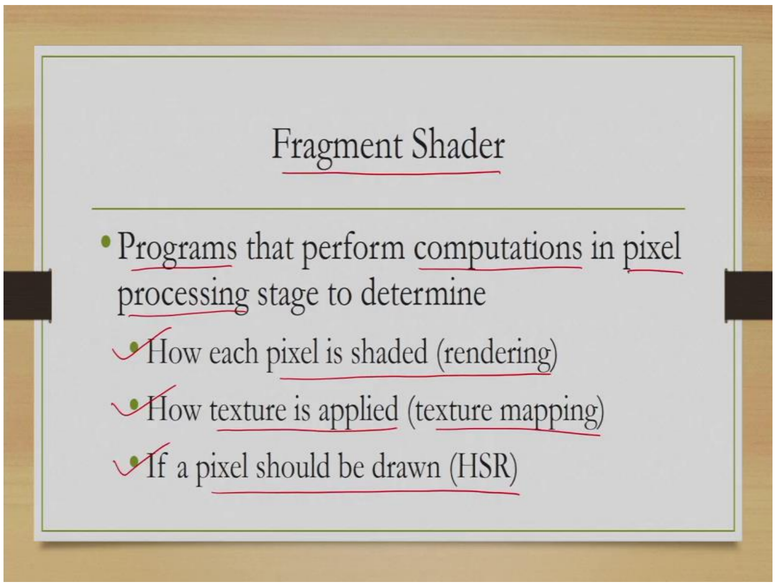


Vertex Shader

- Used to process vertices (i.e., geometry)
  - Modeling transformations
  - Lighting
  - Projection to screen coordinates

In case of vertex shader, what happens is that, these programs are used to process vertices or the geometry. Essentially these programmes are used to perform modeling transformations, lighting and projection to screen coordinates which involve all the intermediate transformations of view transformation. And conceptually the window to view put transformation as well.

(Refer Slide Time: 30:52)



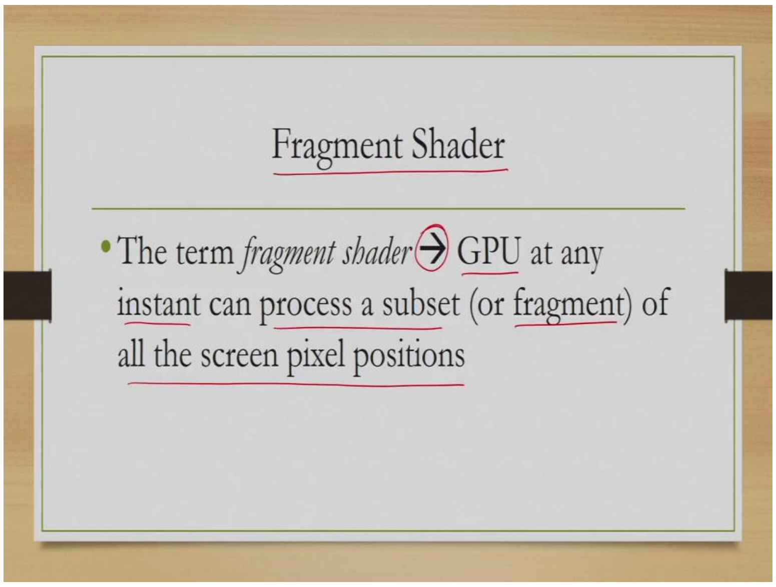
Fragment Shader

- Programs that perform computations in pixel processing stage to determine
  - How each pixel is shaded (rendering)
  - How texture is applied (texture mapping)
  - If a pixel should be drawn (HSR)

In case of fragment shader it does a different job, these are programmes that perform the computations required for pixel processing. Now what are the computations, those are related to how each pixel is rendered, how texture is applied or texture mapping and whether to draw a pixel or not that is hidden surface removal. So, these 3 are the task done by fragment shaders. Note that all these 3 are related to processing of pixels. So, vertex shaders are processing of vertices mostly related to transformations from modeling coordinate to device coordinate, and all the transformations in between.

Whereas fragment shaders deal with pixel processing that is rendering of pixels applying textures as well as performing hidden surface removal at the pixel level.

(Refer Slide Time: 32:14)

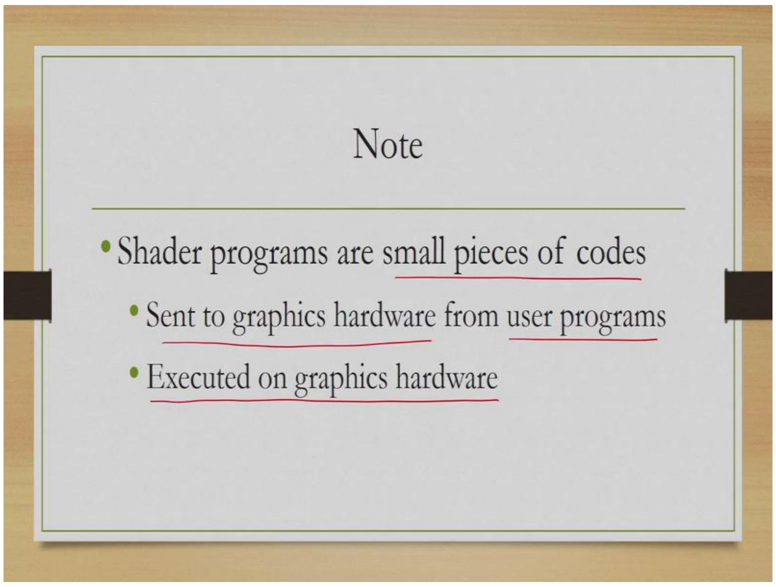


Fragment Shader

- The term *fragment shader* $\rightarrow$ GPU at any instant can process a subset (or fragment) of all the screen pixel positions

Now, why it is called fragment shader the pixel processing units, it implies that GPU at any instant can process a subset or fragment of all the screen pixels that are present. So, at a time a subset of the screen pixels are processed hence it is called a fragment shader.

(Refer Slide Time: 32:43)

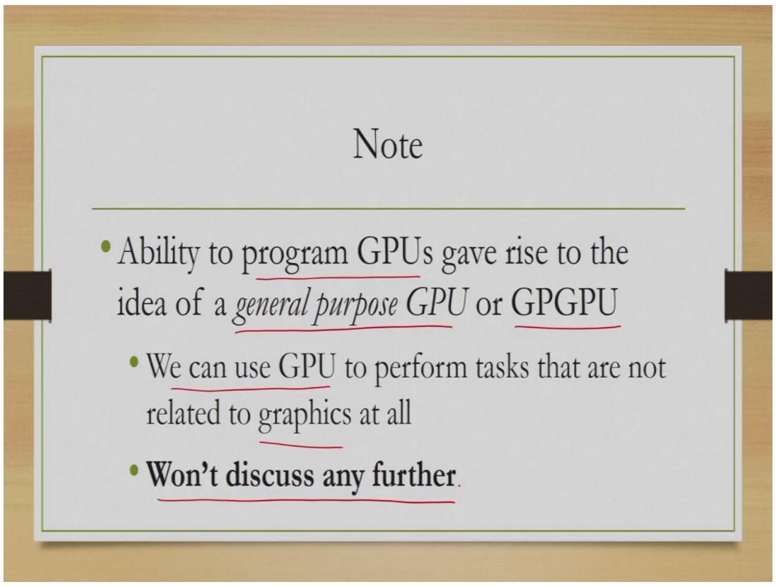


Note

- Shader programs are small pieces of codes
  - Sent to graphics hardware from user programs
  - Executed on graphics hardware

Now, this shader programs are small pieces of codes and they are sent to graphics hardware from user programs so essentially by calling some APIs and executed on graphics hardware. So, we should keep this in mind that they are small pieces of codes which are executed on graphics hardware and which are embedded in user programs, sent to the hardware by the user programs.

(Refer Slide Time: 33:18)

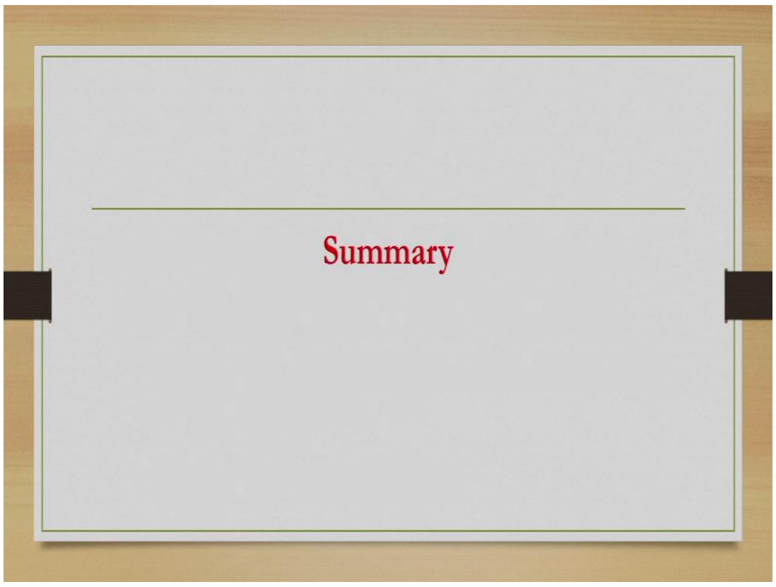


Note

- Ability to program GPUs gave rise to the idea of a *general purpose GPU* or GPGPU
  - We can use GPU to perform tasks that are not related to graphics at all
  - **Won't discuss any further**

In fact, this ability to program GPUs gave rise to a new idea that is the idea of general purpose GPU or GPGPU. Again these are common terms nowadays and you probably have come across this term, this means that we can use GPU for any purpose not necessarily only to perform graphics related operations. So, with the idea of GPGPU we can perform tasks that are not related to graphics at all, however these are very involved subjects and we will not go any further to explain these concepts.

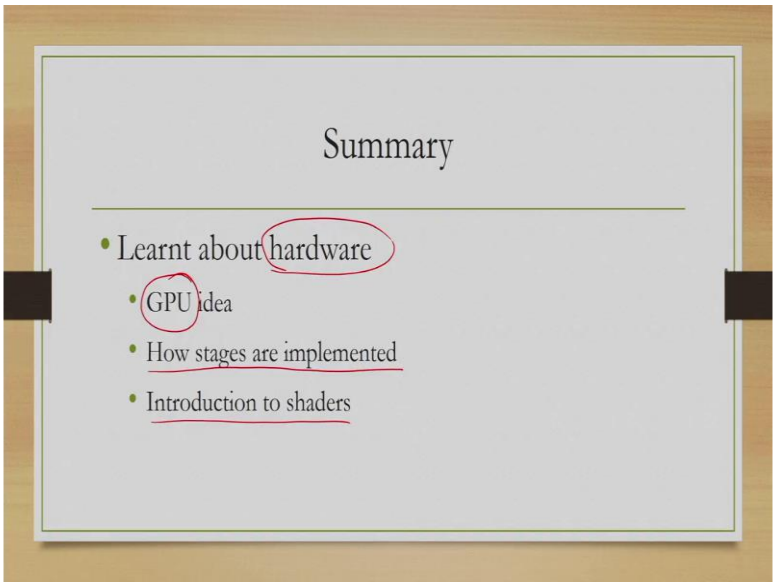
(Refer Slide Time: 34:06)



Summary

So, in summary what we have learnt today, let us try to recap quickly.

(Refer Slide Time: 34:20)

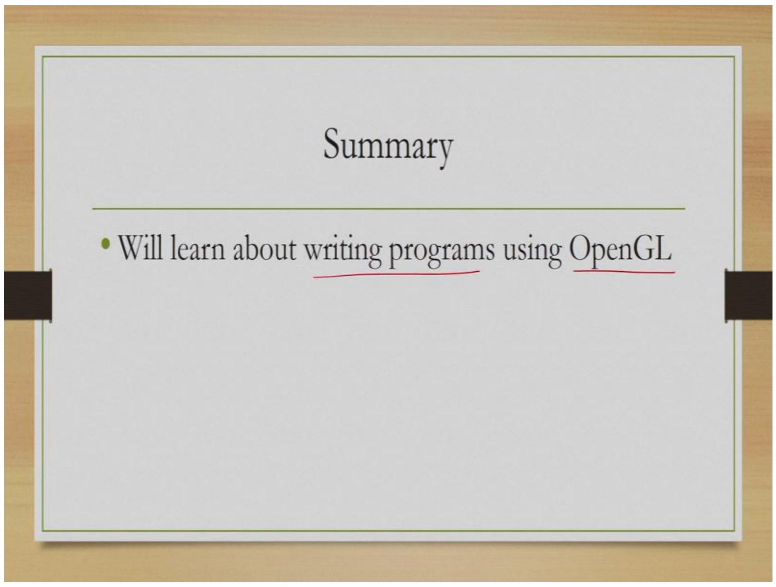


Summary

- Learnt about hardware
  - GPU idea
  - How stages are implemented
  - Introduction to shaders

We learnt about how the hardware works. Now, that means the graphics processing unit which are anyway part of the computer systems that deal with graphics operations. We also learnt how the pipeline stages are implemented in the GPU and got introduced to the idea of shaders and shader programs.

(Refer Slide Time: 35:02)



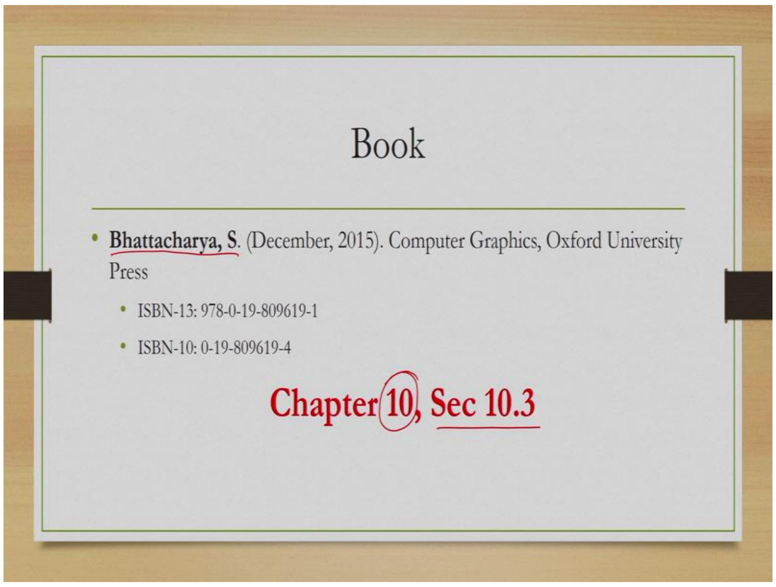
Summary

- Will learn about writing programs using OpenGL

Now, that is about hardware. So, in the previous lecture and today's lecture we learnt about graphics hardware.

We started with discussion on general architecture of a graphics system a very generic architecture, then explain different terms and then in some details learnt how the GPU works. One component remains that is how as a programmer we can write a programme to perform graphics operations. That we will learn that aspect of the course that is writing programs to perform graphics operation or create a scene on the screen that we will learn in the next lecture, where we will learn about writing programs using OpenGL which is a graphics library.

(Refer Slide Time: 36:10)



Book

- Bhattacharya, S. (December, 2015). Computer Graphics, Oxford University Press
  - ISBN-13: 978-0-19-809619-1
  - ISBN-10: 0-19-809619-4

Chapter 10, Sec 10.3

Whatever we have discussed today you can find in this book, specifically Chapter 10, Section 10 point 3. That is all for today, see you in the next lecture. Thank you and good bye.