# 8. Resources and the File System

(part. 2)

Game Player Experience Design

Prof. H. Kang

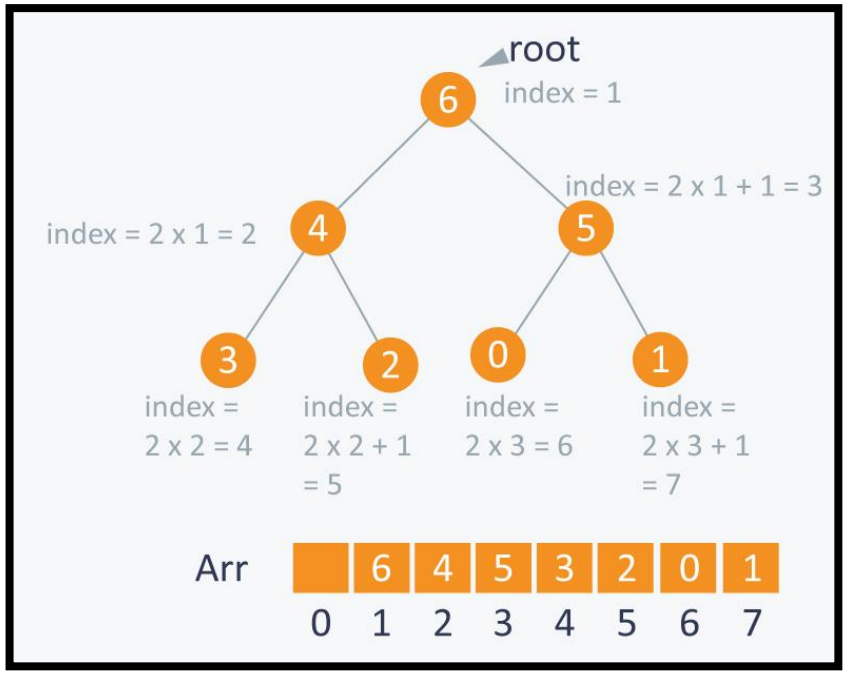
# Memory Management for Resources

## Memory management?

- Resource management is closely related to memory management, because we must inevitably decide where the resources should end up in memory once they have been loaded.
- The design of a game engine's memory allocation subsystem is usually closely tied to that of its resource manager.

## Heap-based resource allocation

- One approach is to simply ignore memory fragmentation issues and use a general-purpose heap allocator to allocate the resources.
- A heap is a specialized tree-based data structure in which all the nodes of the tree are in a specific order.
- Heap-based resource allocation works best if the game is only intended to run on personal computers (since OS supports good virtual memory allocation).

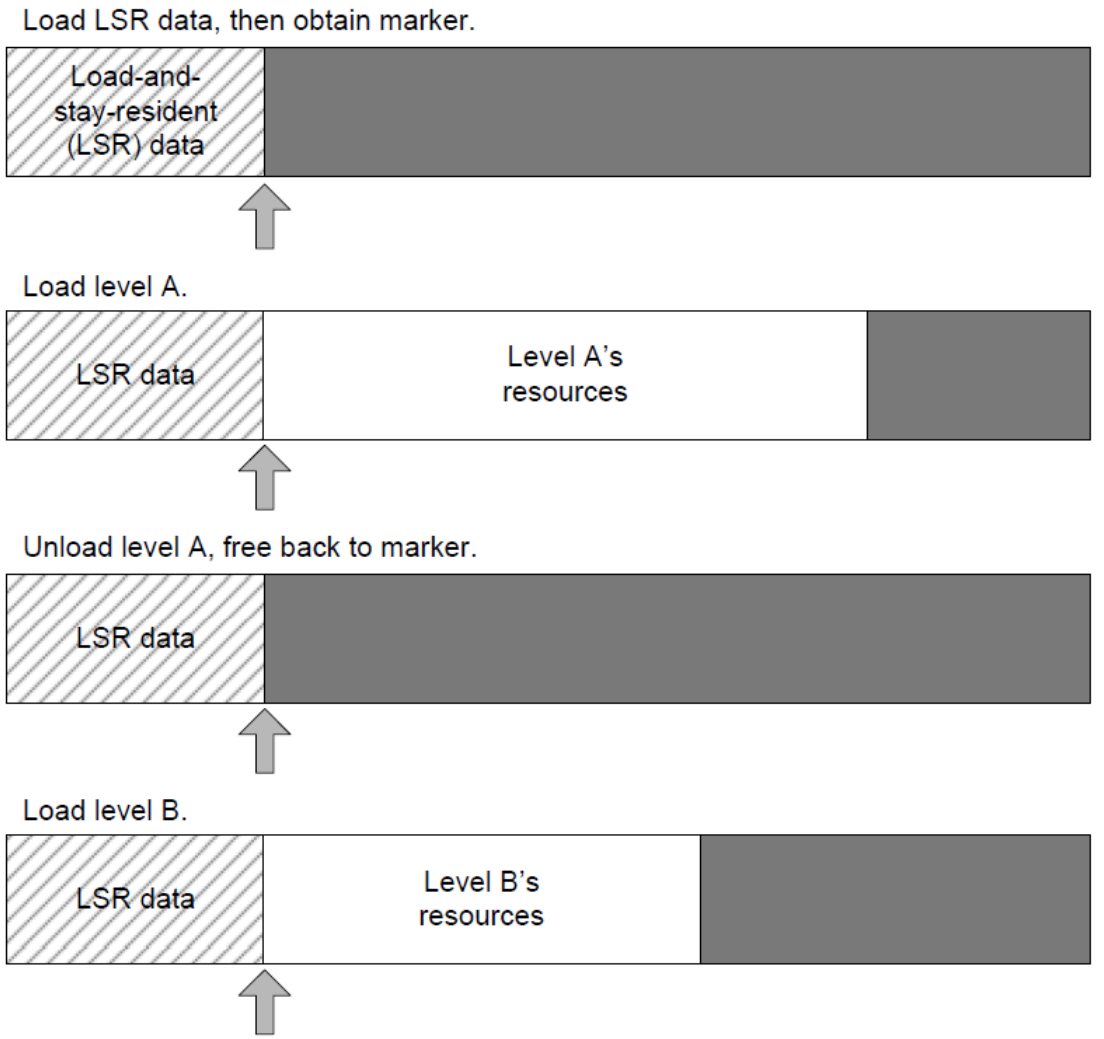



heap[h]

# Memory Management for Resources

## Stack-based resource allocation

- A stack allocator does not suffer from fragmentation problems, because memory is allocated contiguously and freed in an order opposite to that in which it was allocated.
- A stack allocator can be used to load resources if the following two conditions are met:
  - The game is linear and level-centric (i.e., the player watches a loading screen, then plays a level, then watches another loading screen, then plays another level).
  - Each level fits into memory in its entirety.

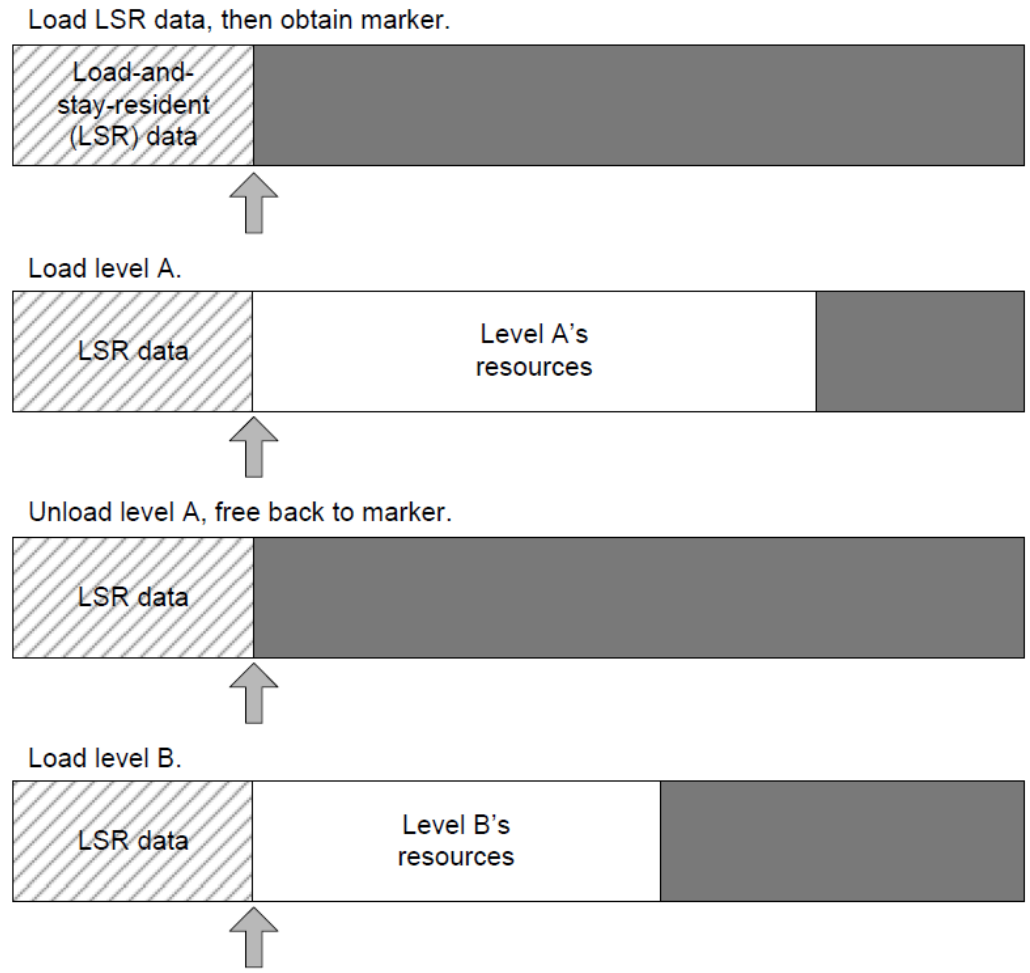



loading resources using a stack allocator

# Memory Management for Resources

## Stack-based resource allocation

- Figure on the right shows the example:
  1) When the game first starts up, the global resources are allocated first.
  2) The top of the stack is then marked, so that we can free back to this position later.
  3) To load a level, we simply allocate its resources on the top of the stack.
  4) When the level is complete, we can simply set the stack top back to the marker we took earlier.
- A double-ended stack allocator can be used to augment this approach.
  - The lower stack can be used for persistent data loads, while the upper can be used for temporary allocations that were freed every frame.

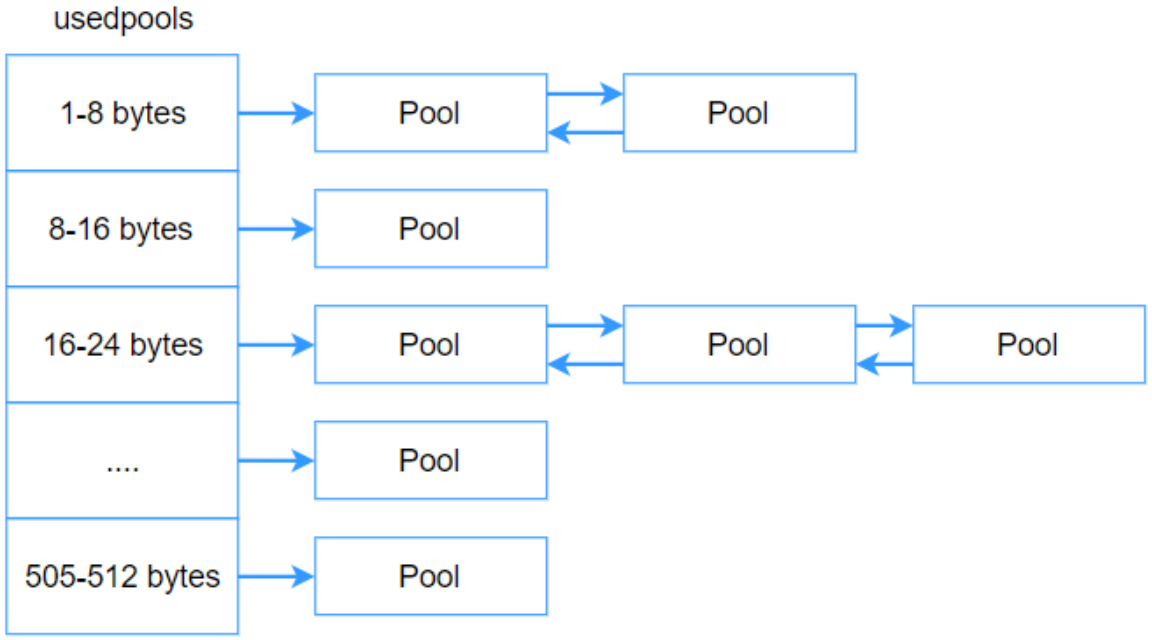



loading resources using a stack allocator

# Memory Management for Resources

## Pool-based resource allocation

- Another resource allocation technique is to load resource data in equally sized chunks.
- Because the chunks are all the same size, they can be allocated using a pool allocator.
  - When resources are later unloaded, the chunks can be freed without causing fragmentation.
- A chunk-based allocation approach requires that all resource data be laid out in a manner that permits division into equally sized chunks.

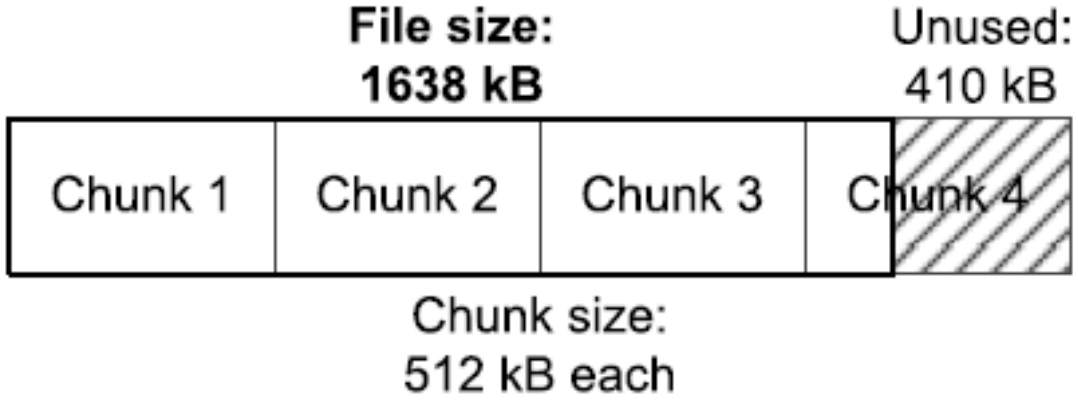



pool[pool]

# Memory Management for Resources

## Pool-based resource allocation

- One big trade-off inherent in a chunky resource allocation scheme is wasted space.
  - Unless a resource file's size is an exact multiple of the chunk size, the last chunk in a file will not be fully utilized.
  - Choosing a smaller chunk size can help to mitigate this problem, but the smaller the chunks, the more onerous the restrictions on the layout of the resource data.
  - A typical chunks are a few kibibytes.




the last chunk of a resource file is often not fully utilized

# Memory Management for Resources

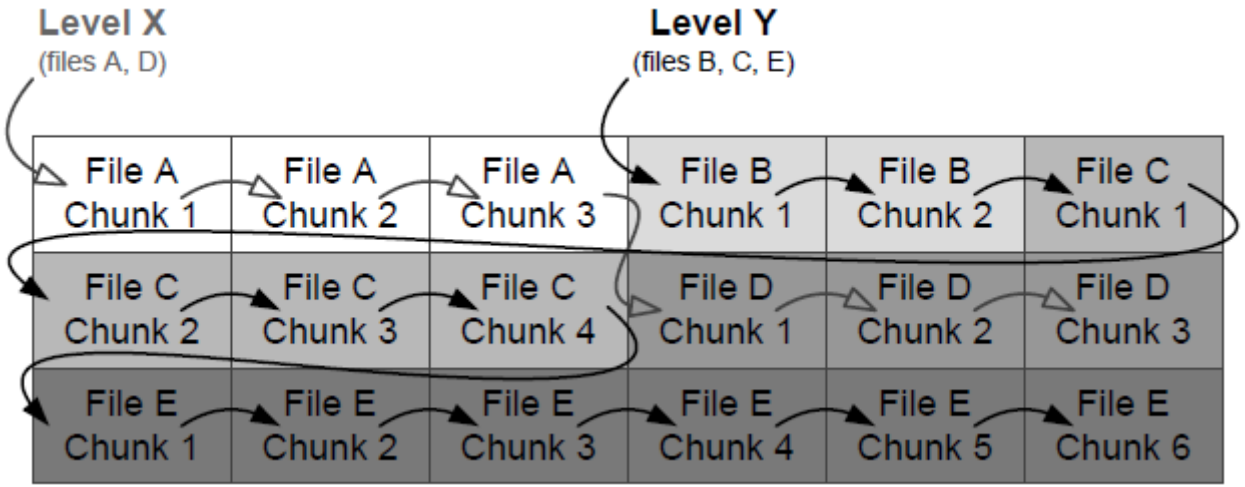
## Resource chunk allocators

- One way to limit the effects of wasted chunk memory is to set up a special memory allocator that can utilize the unused portions of chunks.
- We usually call it a *resource chunk allocator*.
- To implement this, we need to maintain a linked list of all chunks that contain unused memory, along with the locations and sizes of each free block.
- Unfortunately, there's a problem:
  - If we allocate memory in the unused regions of our resource chunks, we cannot free part of a chunk when those chunks are freed.
  - A simple solution to this problem is to only use free-chunk allocator for memory requests whose lifetimes match the lifetime of the level with which a particular chunk is associated.

# Memory Management for Resources

## Pool-based resource allocation

- Each chunk in the pool is typically associated with a particular game level. (One simple way to do this is to give each level a linked list of its chunks.)
- This allows the engine to manage the lifetimes of each chunk appropriately, even when multiple levels with different life spans are in memory concurrently.
- Example
  - When level X is loaded, it might allocate and make use of N chunks.
  - Later, level Y might allocate an additional M chunks.
  - When level X is eventually unloaded, its N chunks are returned to the free pool.
  - If level Y is still active, its M chunks need to remain in memory.

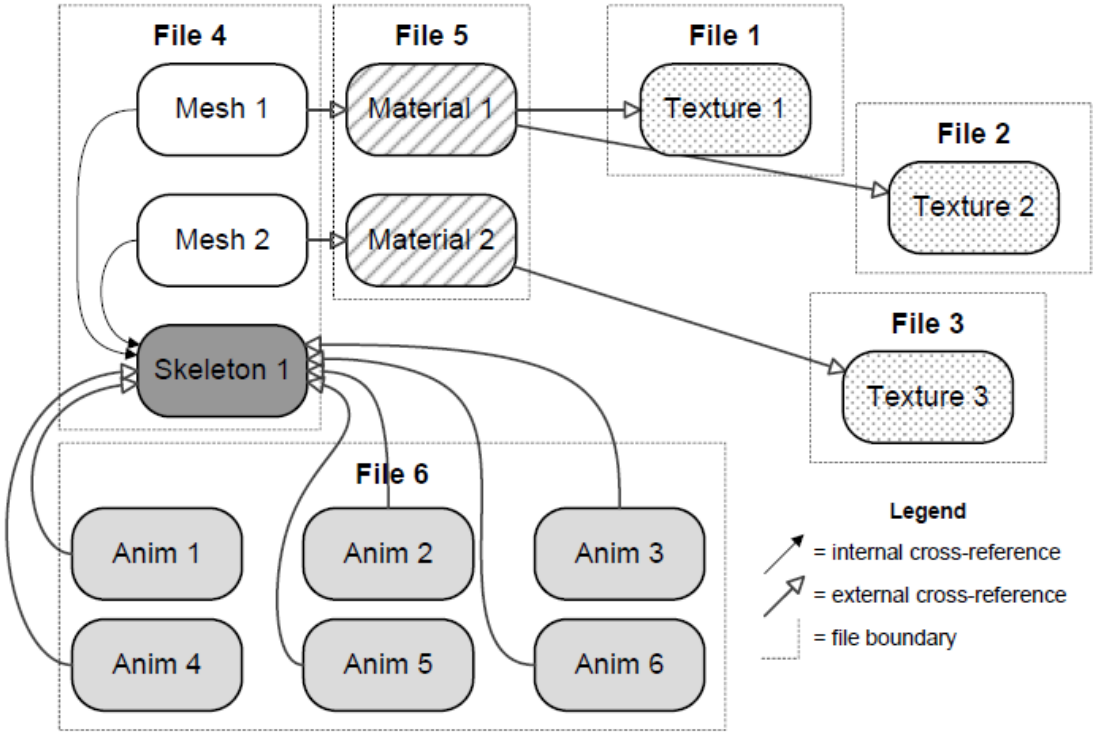



chunky allocation of resources for levels X and Y.

# Cross-reference

## Composite resources and referential integrity

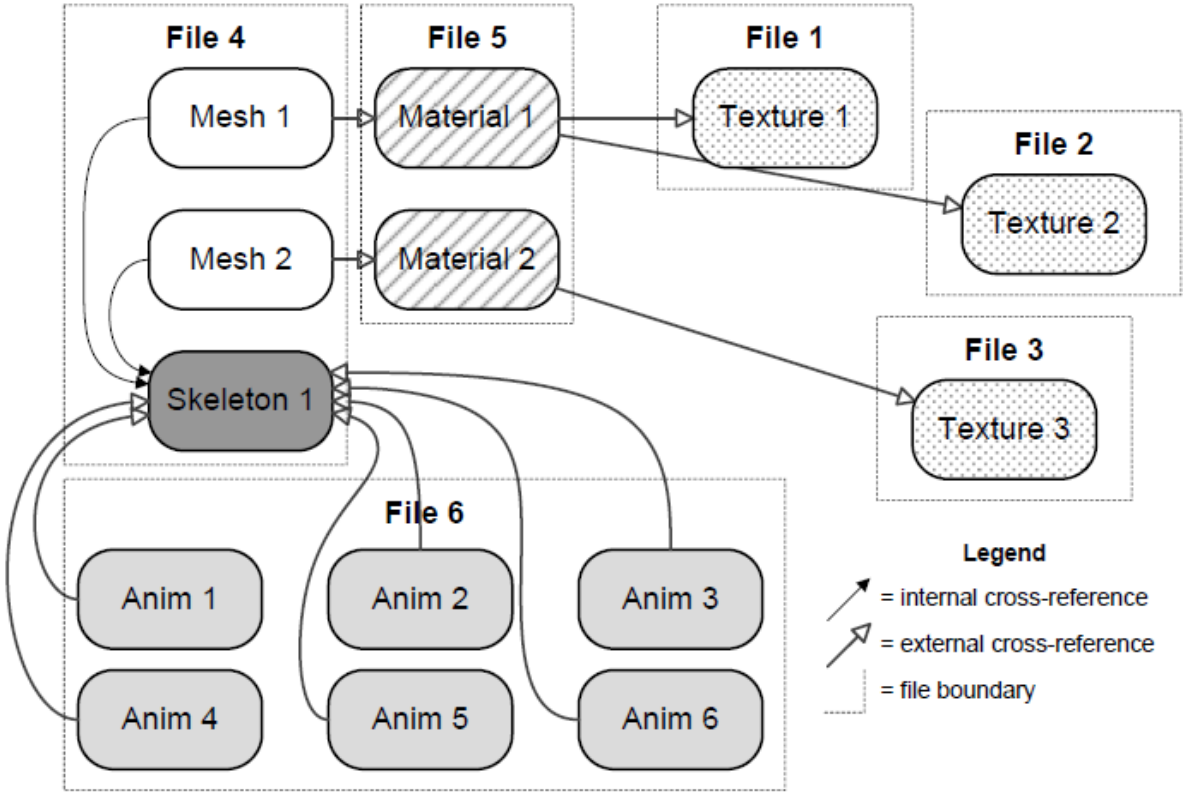
- Usually, a game's resource database consists of multiple resource files, each file containing one or more data objects.
- These data objects can refer to and depend upon one another in arbitrary ways.
  - For example, a mesh data structure might contain a reference to its material, which in turn contains a list of references to textures.



# Cross-reference

## Composite resources and referential integrity

- Usually cross-references imply dependency
  - A reference between two objects within a single file is *internal cross-reference*.
  - A reference between two objects in a different file is *external cross-reference*.
- We use the term *composite resource* to describe a self-sufficient cluster of interdependent resources.
  - A model is a composite resource consisting of one or more triangle meshes, an optional skeleton and an optional collection of animations.
  - To fully load a composite resource like a 3D model into memory, all of its dependent resources must be loaded as well.



# Cross-reference

## Handling cross-references between resources

- One of the more-challenging aspects of implementing a resource manager is managing the cross-references between resource objects and guaranteeing that referential integrity is maintained.
- In C++, a cross-reference between two data objects is usually implemented via a *pointer* or a *reference*. However, pointers are just memory addresses – they lose their meaning when taken out of the context of the running application.
- One good approach is to store each cross-reference as a string or hash code containing the unique id of the referenced object.
  - In this case, every resource object must have a globally unique identifier (GUID).
  - The runtime resource manager maintains a global resource look-up table.
  - Whenever a resource object is loaded into memory, a pointer to that object is stored in the table with its GUID as the look-up key.
  - After all resource objects have been loaded into memory, a pass over all of the objects can be created by looking up the address of each cross-referenced object.

# Cross-reference

## Handling cross-references between resources

- Another approach that is often used when storing data objects into a binary file is to convert the *pointers* into file *offsets*.
  - Consider a group of C or C++ objects that cross-reference each other via pointers.
  - To store this group of objects into a binary file, we need to visit each object once (and only once) in an arbitrary order and write each object's memory image into the file sequentially (serialized data).

resource to binary file

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | ... |
| 1 | 0 | 1 | 0 | ... |
| 1 | 1 | 1 | 1 | ... |
| 0 | 0 | 0 | 1 | ... |
| ... | ... | ... | ... | ... |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | ... |
| 1 | 1 | 1 | 0 | ... |
| 1 | 0 | 1 | 0 | ... |
| 0 | 1 | 0 | 1 | ... |
| ... | ... | ... | ... | ... |

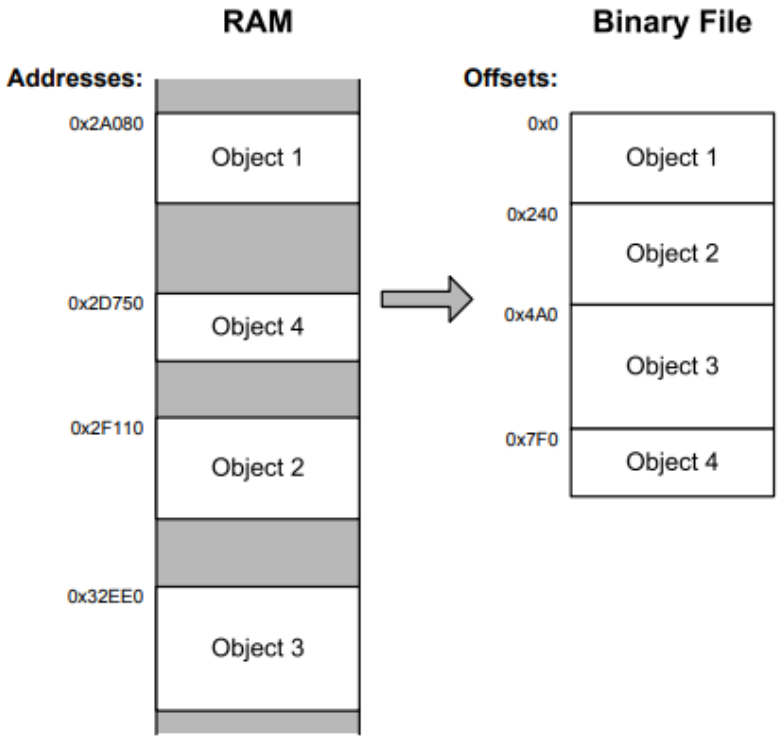
serialize

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | ... |

# Cross-reference

## Handling cross-references between resources

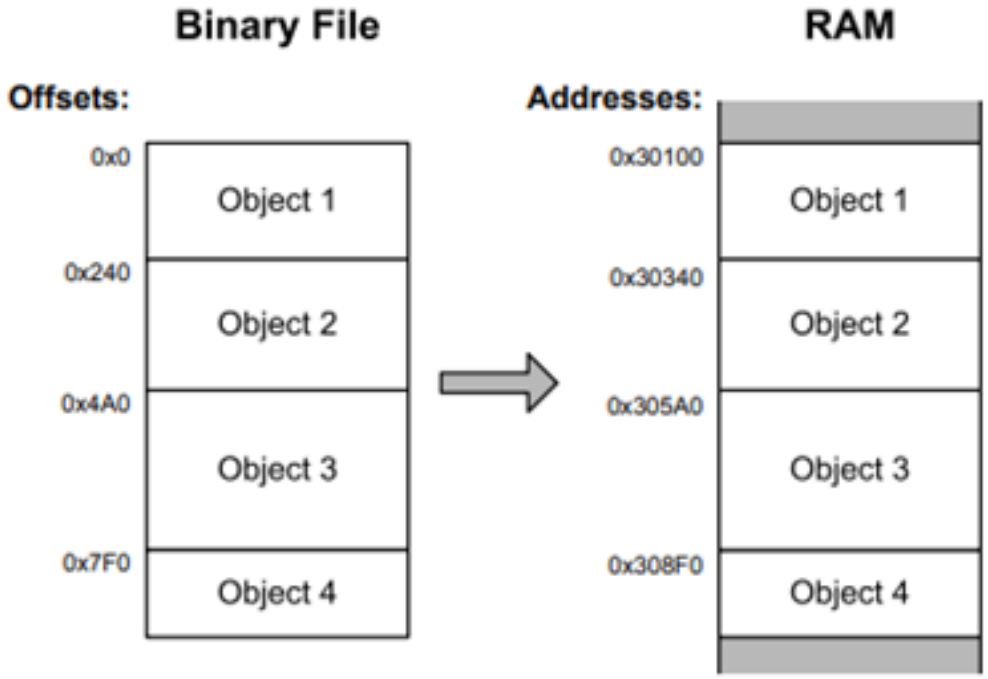
- Another approach that ... (Cont.).
  - This has the effect of serializing the objects into a contiguous image within the file, even when their memory images are not contiguous in RAM.
  - Because the objects' memory images are now *contiguous* within the file, we can determine the offset of each object's image relative to the beginning of the file.
  - During the process of writing the binary file image, we locate every pointer within every data object, convert each pointer into an offset and store those offsets into the file in place of the pointers.



# Cross-reference

## Handling cross-references between resources

- Another approach that ... (Cont.).
  - When the file's binary image is loaded, the objects contained in the image retain their contiguous layout, so it is trivial to convert an offset into a pointer. We merely add the offset to the address of the file image as a whole.



# Post-load Initialization

## Post-load initialization

- Ideally, each and every resource would be completely prepared by offline tools, so that it is ready for use the moment it has been loaded into memory.
- Practically, this is not always possible. Many types of resources require some “massaging” after having been loaded in order to prepare them for use by the engine.
- The term *post-load initialization* refers to any processing of resource data after it has been loaded.

# Post-load Initialization

## Post-load initialization

- Post-load initialization generally comes in one of two varieties:
  - In some cases, post-load initialization is an unavoidable step. For example, on a PC, the vertices and indices that describe a 3D mesh are loaded into main RAM, but they must be transferred into GPU RAM before they can be rendered. This can only be accomplished at runtime, by creating a Direct X vertex buffer or index buffer, locking it, copying or reading the data into the buffer and then unlocking it.
  - In other cases, the processing done during post-load initialization is avoidable (i.e., could be moved into the tools), but is done for convenience or expedience. For example, a programmer might want to add the calculation of accurate arc lengths to engine's spline library. Rather than spend the time to modify the tools to generate the arc length data, the programmer might simply calculate it at runtime during post-load initialization. Later, when the calculations are perfected, this code can be moved into the tools, thereby avoiding the cost of doing the calculations at runtime.

# GPU data transfer

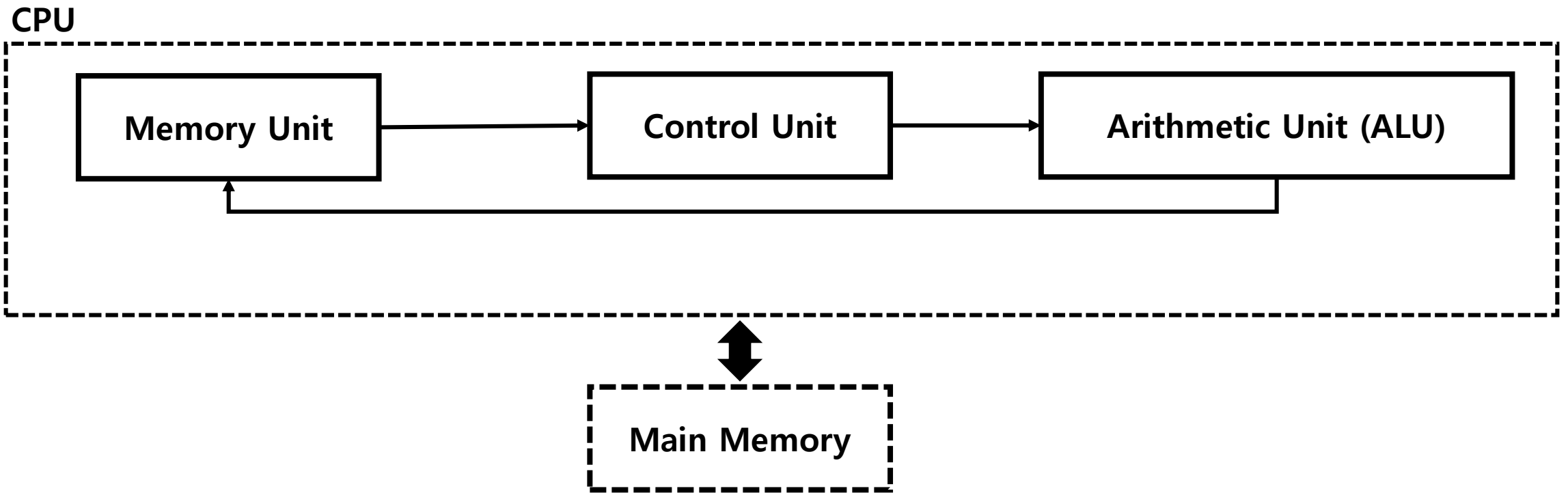
## Resources for GPU

- Some resources should be transmitted to the GPU since they will be processed by the GPU.
- Game engine is run on the CPU-based hardware (e.g. personal computer)
- Therefore, transmitting the resource to the GPU requires additional steps.
- However, they are super difference in terms of architecture and this often causes unexpected performance issues when designing transmitting APIs.
- To understand this step, we must know the CPU and GPU.

# GPU data transfer

## CPU and Memory

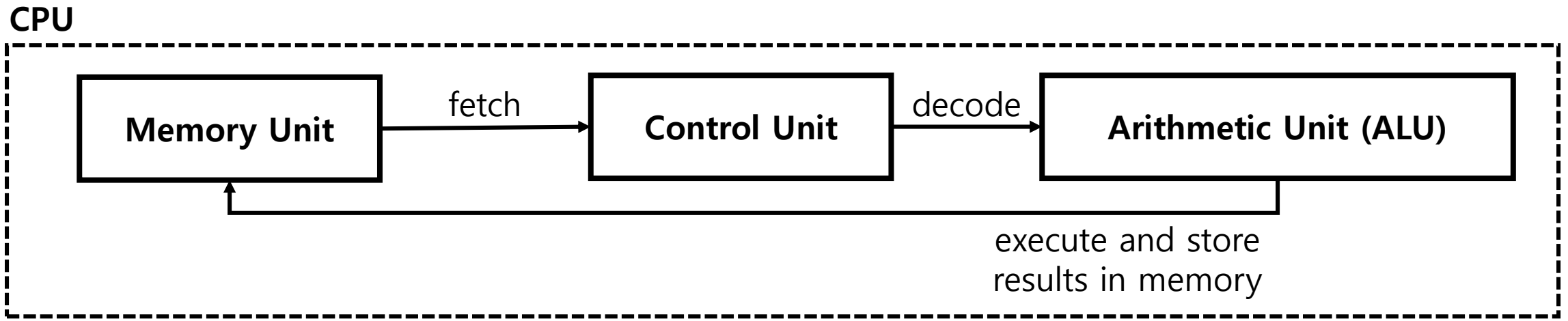
- In personal computer, CPU and memory is the core hardware that defines a computer.
- CPU executes programs.
- Memory stores program operations and data while a program is being executed. There are several types of memory, including register, cache, RAM, virtual memory, etc.



# GPU data transfer

## CPU and Memory

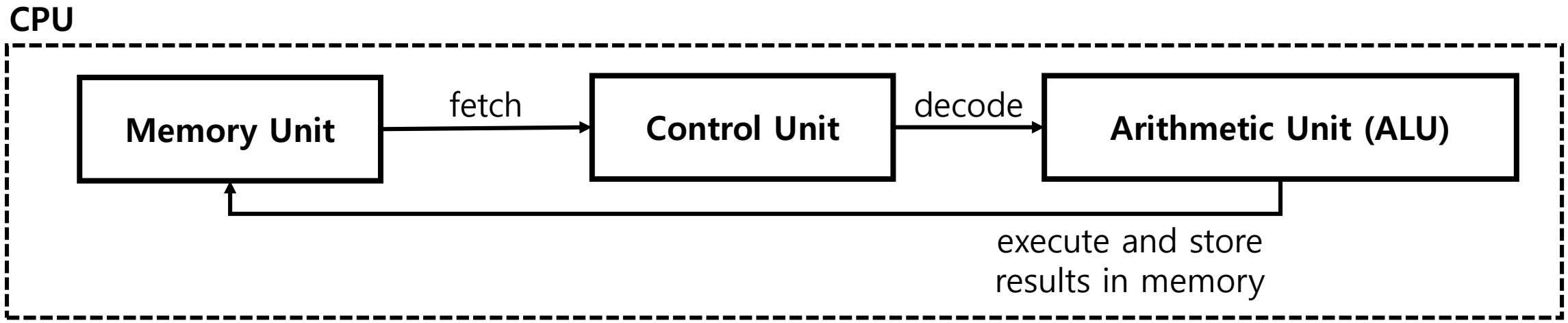
- CPU executes programs using the instruction cycle (known as fetch-decode-execute cycle)
  - **Fetch stage**: the next instruction is fetched from the memory address that is currently stored in the program counter and stored into the instruction register. At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.
  - **Decode stage**: During this stage, the encoded instruction presented in the instruction register is interpreted by the decoder.



# GPU data transfer

## CPU and Memory

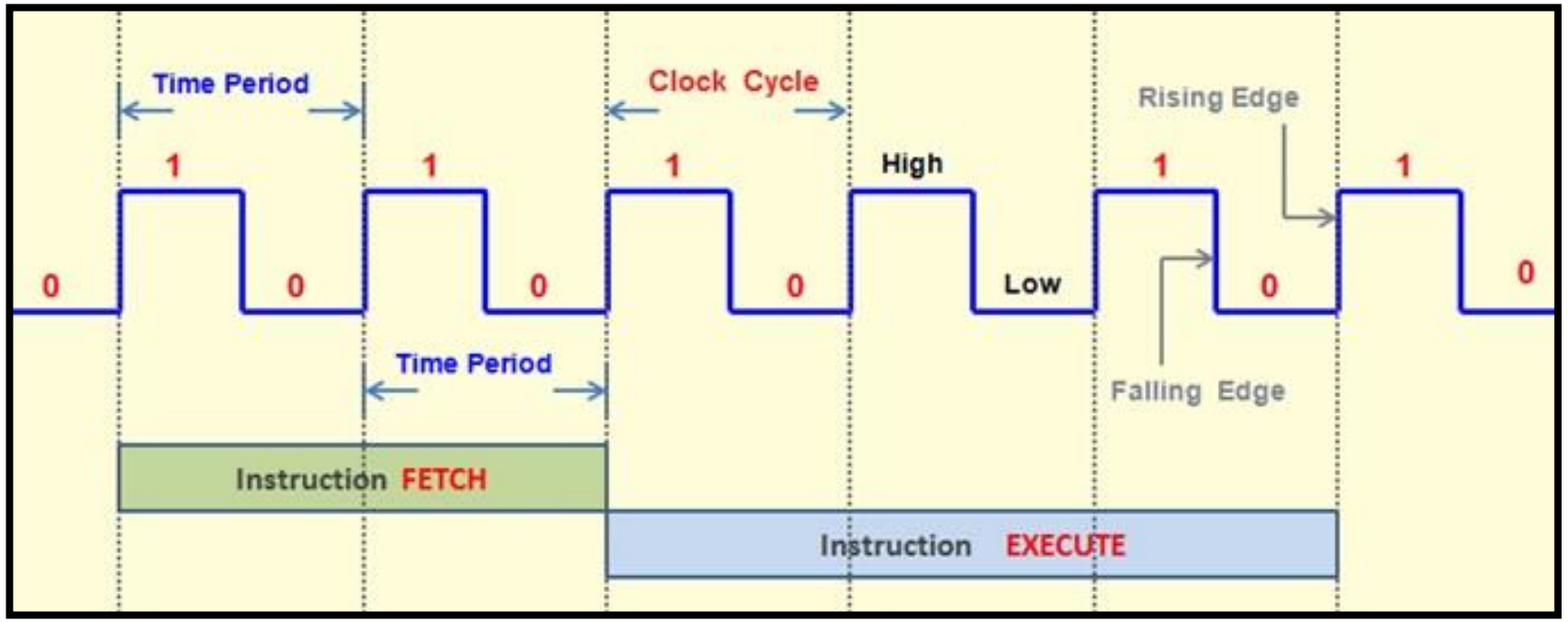
- CPU executes programs ... (Cont.)
  - **Execute Stage**: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant functional units of the CPU to perform the actions required by the instruction, such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register.
  - If the ALU is involved, it sends a condition signal back to the control unit. The result generated by the operation is stored in the memory or sent to an output device.



# GPU data transfer

## Clock speed and computational performance

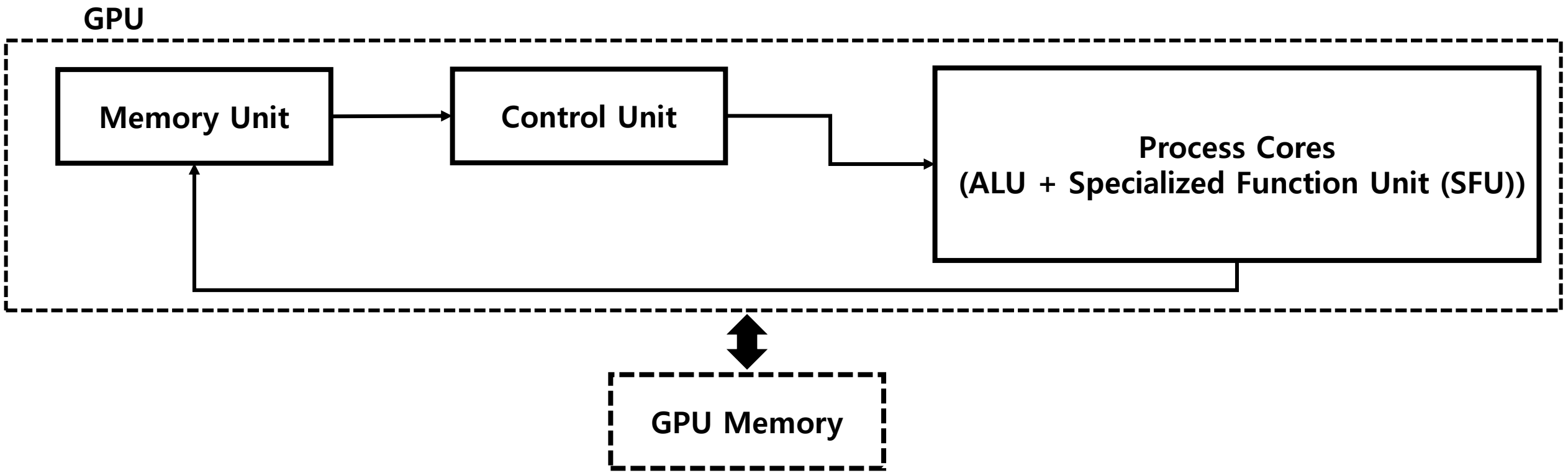
- The performance of CPU has a major impact on the speed at which programs load and how smoothly they run.
- In general, the higher clock speed, more instructions it can process in a time.
- Therefore, CPU is often composed of just a few cores (but has really good core) with lots of cache memory that can handle multiple software threads at a time.



# GPU data transfer

## GPU and Memory

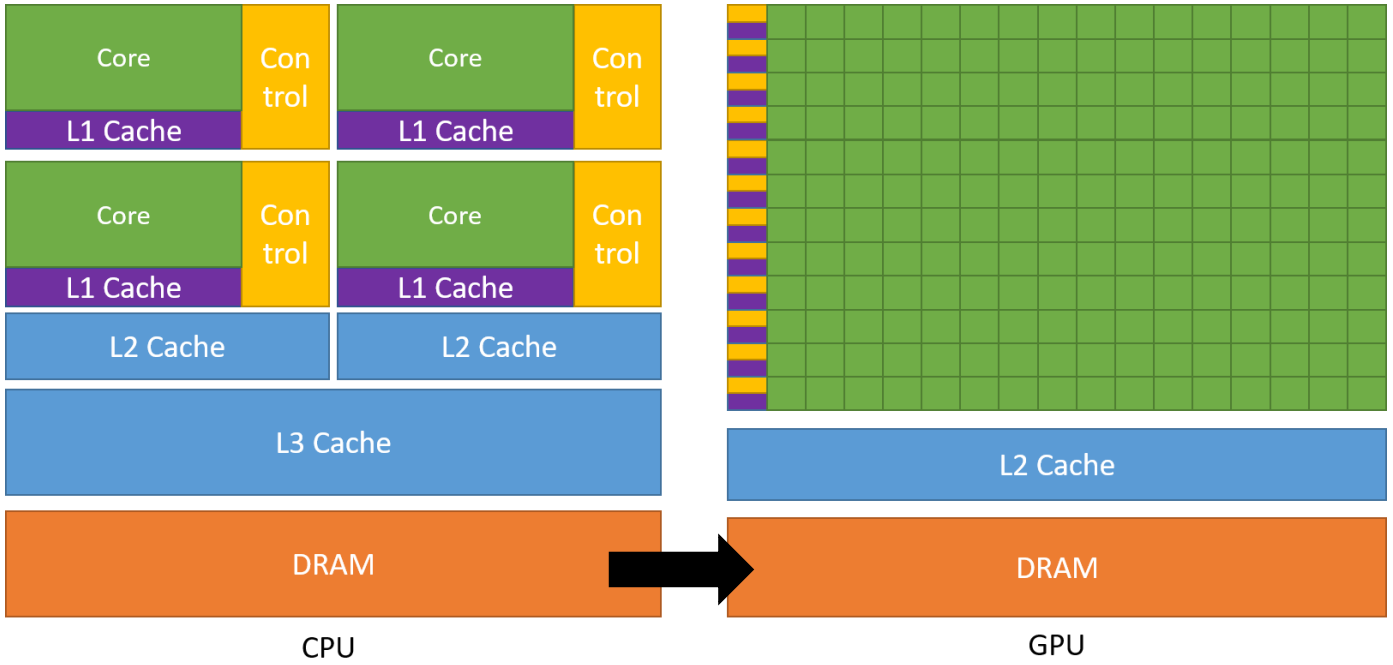
- A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate computer graphics and image processing.
- In contrast to the general-purpose central processing units (CPUs), GPU can process large blocks of data in parallel.
- Therefore, GPU often includes a lot of cores (ALU and SFU)



# GPU data transfer

## CPU – GPU data transfer

- The data transfer between CPU – GPU is very slow.
- Therefore, unnecessary data should not be transmitted to the GPU (considering rendering time, computational time, parallel computing,  CPU/GPU memory, etc.).
- Since GPU is specialized for processing images, instruction strategies should be determined in the CPU stage.
- Therefore, GPU coding often includes CPU coding to handle how the data should be processed in the GPU.




Memory to memory data transfer

# GPU data transfer

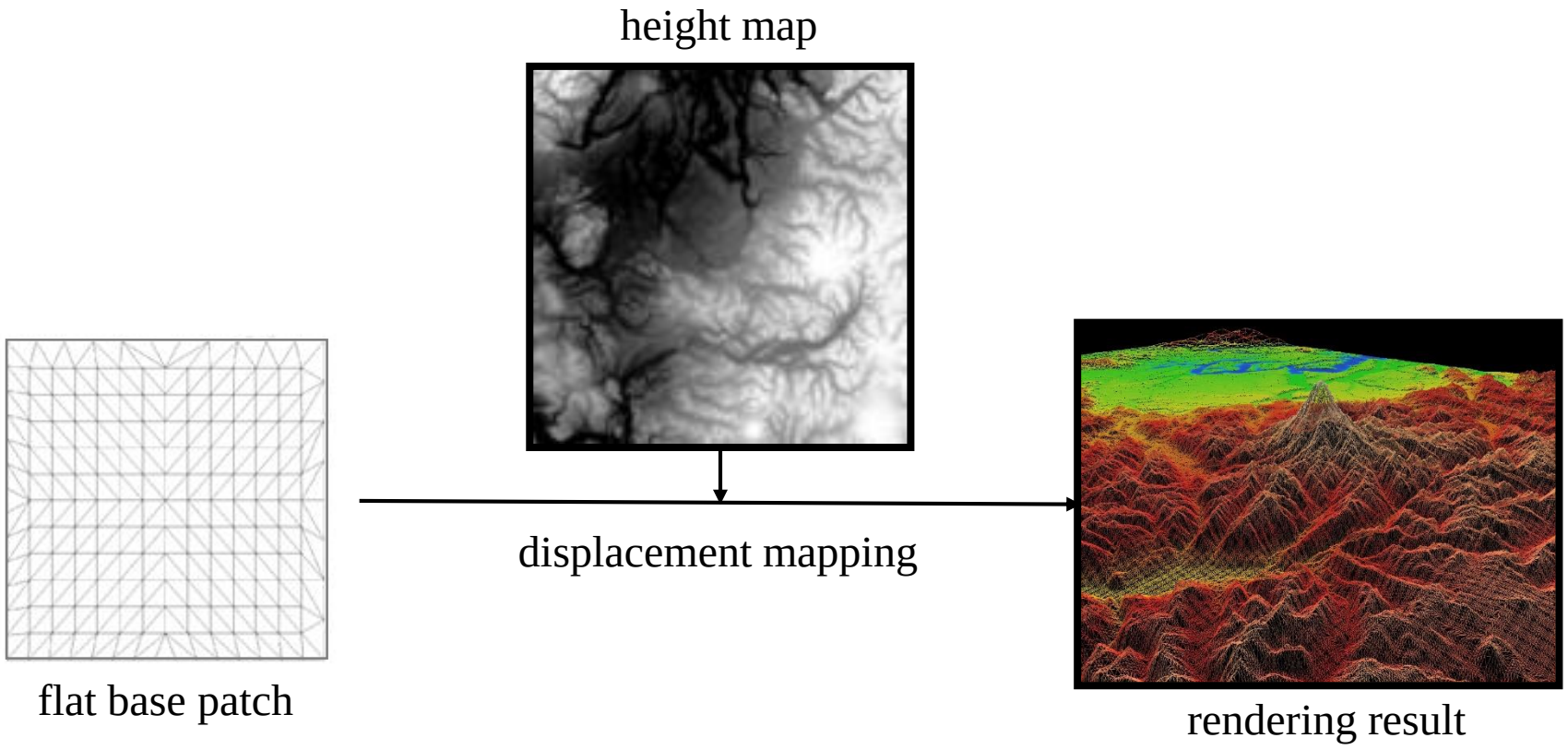
CPU – GPU data transfer

Procedural generation? Copy & Paste?

# GPU data transfer

## Terrain Rendering

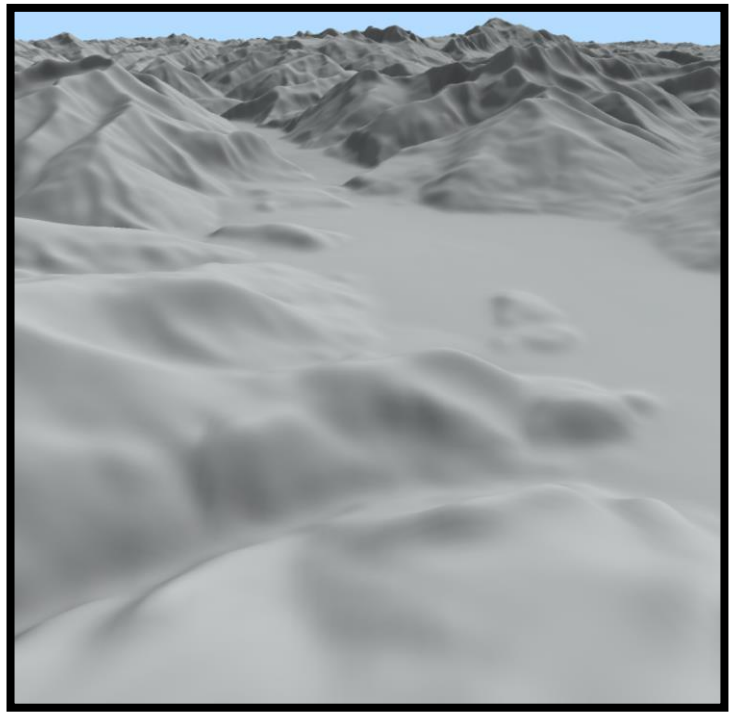
- Terrain rendering is essential for many applications.
- To render the terrain surface, height map based technique is often adopted.
- A heightmap (or elevation map) is a set of height data sampled in a regular grid.
- Each pixel stores surface elevation data .

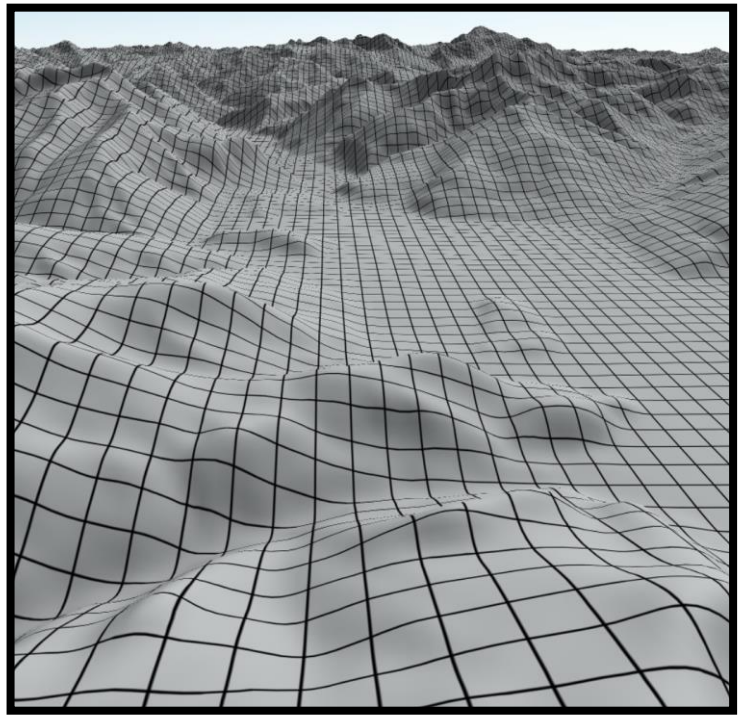


# GPU data transfer

## Terrain Rendering

- Height map samples (or stores) height data with a regular interval.
- Therefore, it merely approximately reconstructs the terrain surface.
- Nonetheless, artifacts are rarely perceived by the average users, and heightmap based techniques are widely accepted in 3D applications.





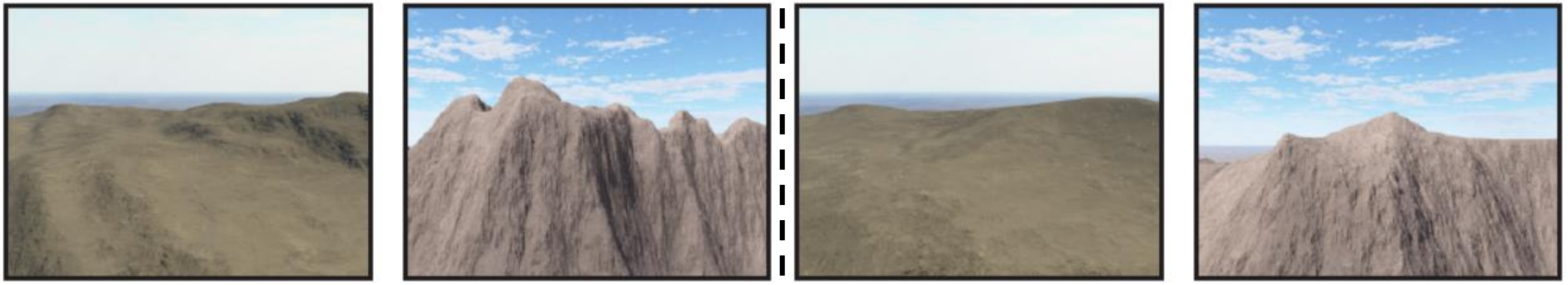
# GPU data transfer

## Terrain Rendering

- High-resolution terrain mesh requires high-resolution height map.
- However, high-resolution height map cannot be loaded to the GPU memory rapidly.
- The use of low-resolution terrain mesh reveals another problem, ugly artifact.

high-resolution terrain mesh

low-resolution terrain mesh

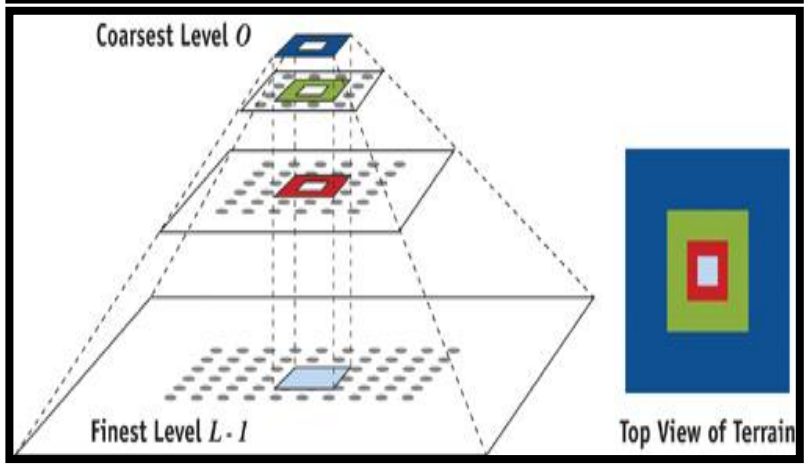


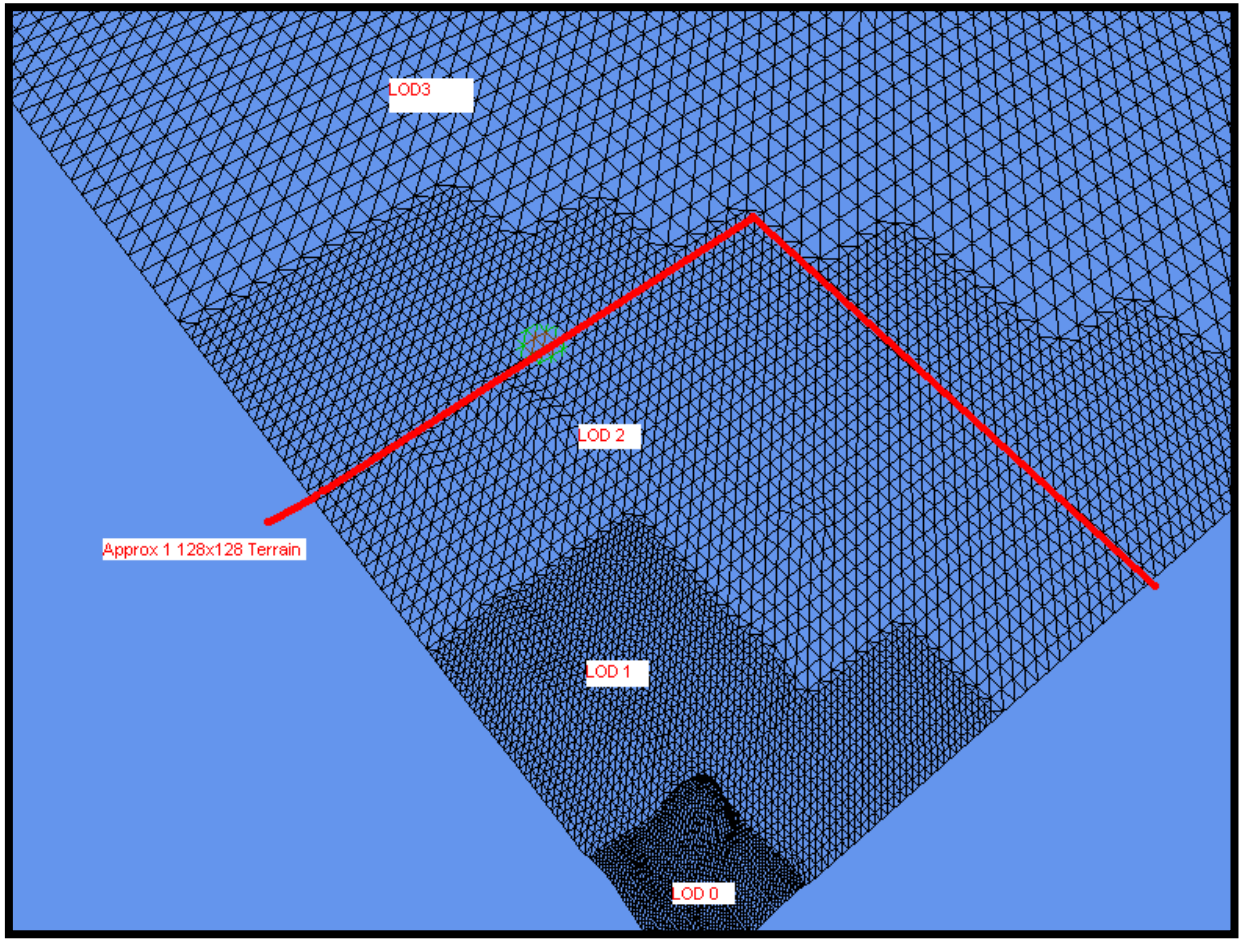
the same resolution height mesh is used

# GPU data transfer

## Terrain Rendering

- To resolve this problem, many techniques have been proposed: procedural detail generation, clip map, level-of-detail, etc.



# Reference

[h] https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/tutorial/#:~:text=A%20heap%20is%20a%20tree,be%20followed%20across%20the%20tree.
[pool] https://csrgxtu.github.io/2020/02/11/How-Python-manage-memory/