The Art of Graphics Programming

Week 1: *Technical Context*

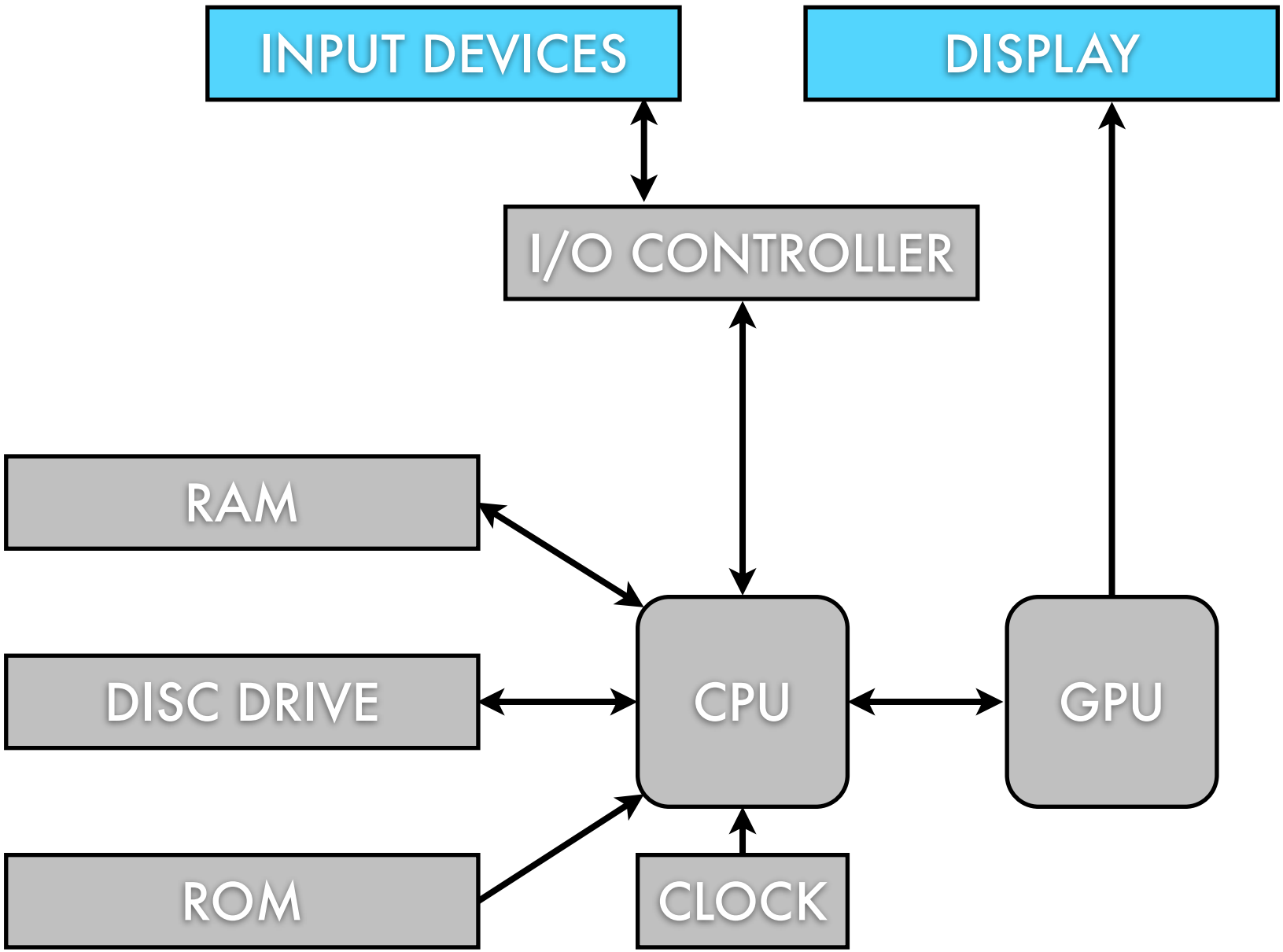
Patrick Hebron
NYU ITP - Fall 2012

Basic Anatomy

CPUs, GPUs, Programming Languages & APIs

Hardware Component Classifications

Hardware Anatomy (Simplified)



A Central Processing Unit (CPU)

is a large-scale integrated circuit, capable of performing the basic arithmetical, logical and input/output calculations required by a computer system.

The CPU serves as the primary clearing house for instructions assigned by the OS and other applications.

As Eric Rosenthal says,
"The CPU is the CPU because it's the CPU."

Examples: Intel i7 Series and AMD Opteron Series

A Graphics Processing Unit (GPU)

is a specialized integrated circuit, designed for the accelerated handling of floating point (decimal) calculations as well as the construction and manipulation of geometric and textual data related to graphics display.

A GPU is also accelerated through its use of numerous parallel computational nodes.

Examples: NVidia GeForce Series and ATI Radeon Series

Random-Access Memory (RAM)

is a type of computer memory storage hardware
that uses a solid-state medium to facilitate
the rapid retrieval and writing of data.

RAM is designed in such a way that any memory unit can
be accessed as quickly as any other. However, RAM is *volatile,*
meaning that its memory state is lost when power is removed.

RAM is generally used for the temporary storage of data
required by the runtime processes of the OS and other software.

A Disc Drive

is a type of computer memory storage hardware used for the persistent storage of large quantities of data.

Until recently, this term would generally imply a magnetic spinning-platter medium. In the past few years, high-speed solid-state drives have become increasingly prevalent.

Disc drives are slower than RAM. However, they are less expensive per memory unit and are also non-volatile.

Software Component Classifications

An Operating System

is a collection of software that manages a computer's hardware components and provides services such as event handing and memory allocation to programs running within the OS.

Modern operating systems often provide additional APIs such as windowing environments, user interface widgets and network protocols that assist third-party development for the OS.

Examples: Mac OS, Windows, Linux, iOS and Android

A Virtual Machine

is an operating system that runs within another operating system.

This allows several user terminals to be hosted on a single hardware system or for a single terminal to draw upon the resources of multiple hardware systems.

Virtual machines are also used to simplify the process of creating cross-platform software. For example, software written in Java and run within the JVM will behave almost identically on Mac, Windows, Linux and Solaris computers.

Examples: Java Virtual Machine (JVM), VMware and Parallels

A Programming Language

is a human-readable vocabulary and syntax, which is used to define logical instructions that can be interpreted and executed by a computer.

This term generally implies a language that is *Turing complete*, meaning that it is capable of representing any deterministic calculation whatsoever.

Examples: C, C++, Objective-C, Java and Python

A Graphics API

provides a series of tools related to the creation,
manipulation and realtime rendering of 2D or 3D geometries.

Due to the computational volume of this task,
specialized hardware (the GPU) is
commonly used to accelerate the process.

A graphics API provides an interface for and manages the
relationship between the software and hardware
components of a realtime rendering pipeline.

Examples: OpenGL and DirectX

An Application Framework

mediates the low-level operations of a
programming language and additional APIs in order
to provide developers with easier access to
functions that would be commonly
used in the development of software of a particular kind.

An application framework is not a programming language, per se.
Instead, it can be seen as a shorthand vocabulary
that encapsulates or reduces a larger, more generalized
and often more complex vocabulary.

Examples: Processing, OpenFrameworks, Cinder, Pocode and Polycode

A Shader Language

exists within a Graphics API and provides a syntax and vocabulary for the expression of logical instructions specifically related to the capabilities of a graphics card's specialized design.

In general, shaders are used to define and augment the material properties of computer-generated surfaces. These properties include a surface's perspectival distortion, illumination and texturing.

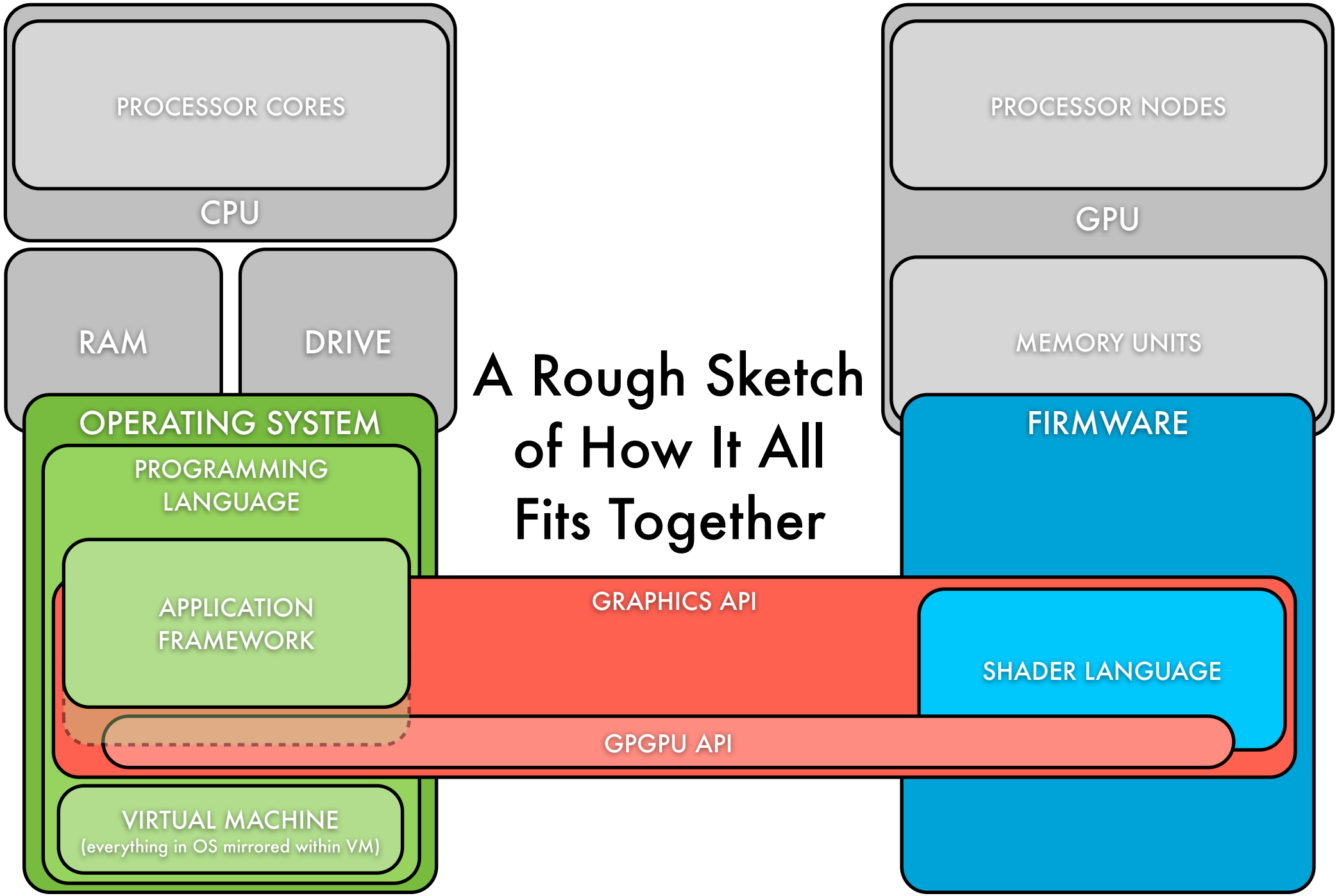
Examples: GLSL, Cg, HLSL and RenderMan

A GPGPU API

or, *General Purpose* GPU API, provides an interface for performing non-graphics calculations through the high-performance, specialized hardware of a graphics card.

Examples: OpenCL and CUDA

Hardware and Software as an Integrated System


PROCESSOR CORES
CPU
PROCESSOR NODES
GPU
RAM
DRIVE
MEMORY UNITS
A Rough Sketch of How It All Fits Together
OPERATING SYSTEM
PROGRAMMING LANGUAGE
APPLICATION FRAMEWORK
VIRTUAL MACHINE
(everything in OS mirrored within VM)
FIRMWARE
GRAPHICS API
SHADER LANGUAGE
GPGPU API

For optimal performance, an integrated system should distribute work amongst its components in accordance with the strengths and weaknesses of each.

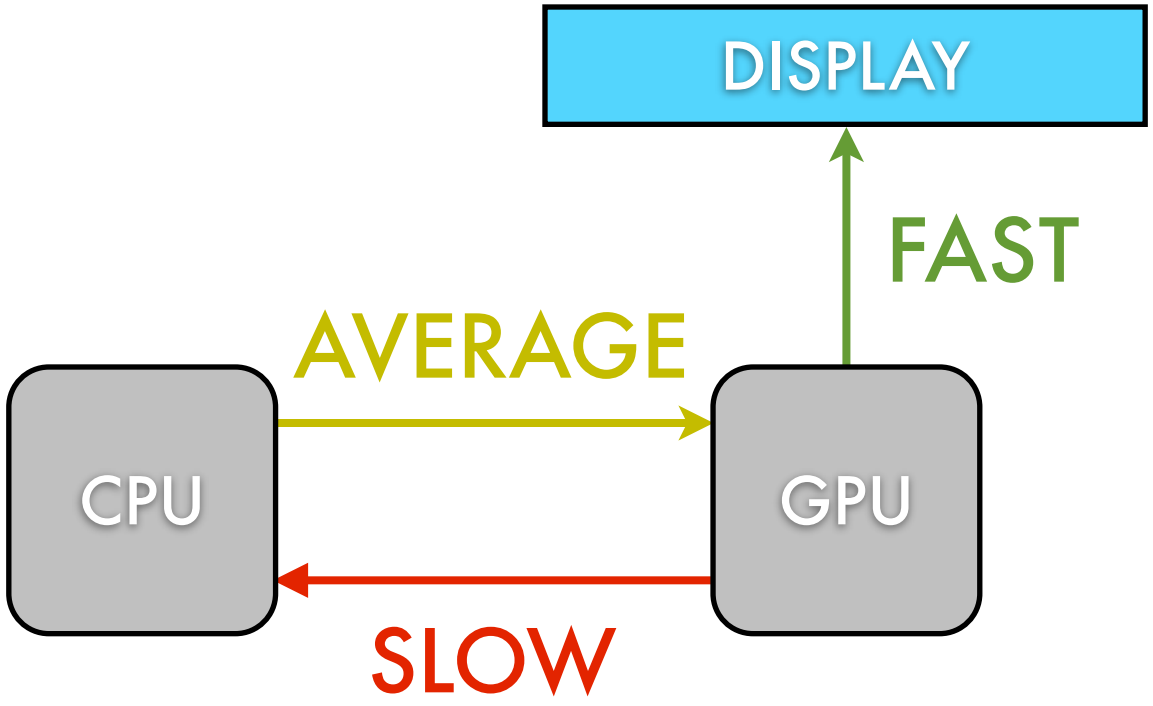
The primary considerations for a real-time graphics pipeline are:

- Computation speed
- Data transfer speed

The GPU is extremely fast, but designed for specific conditions.
The CPU is slower, but designed for all terrains.

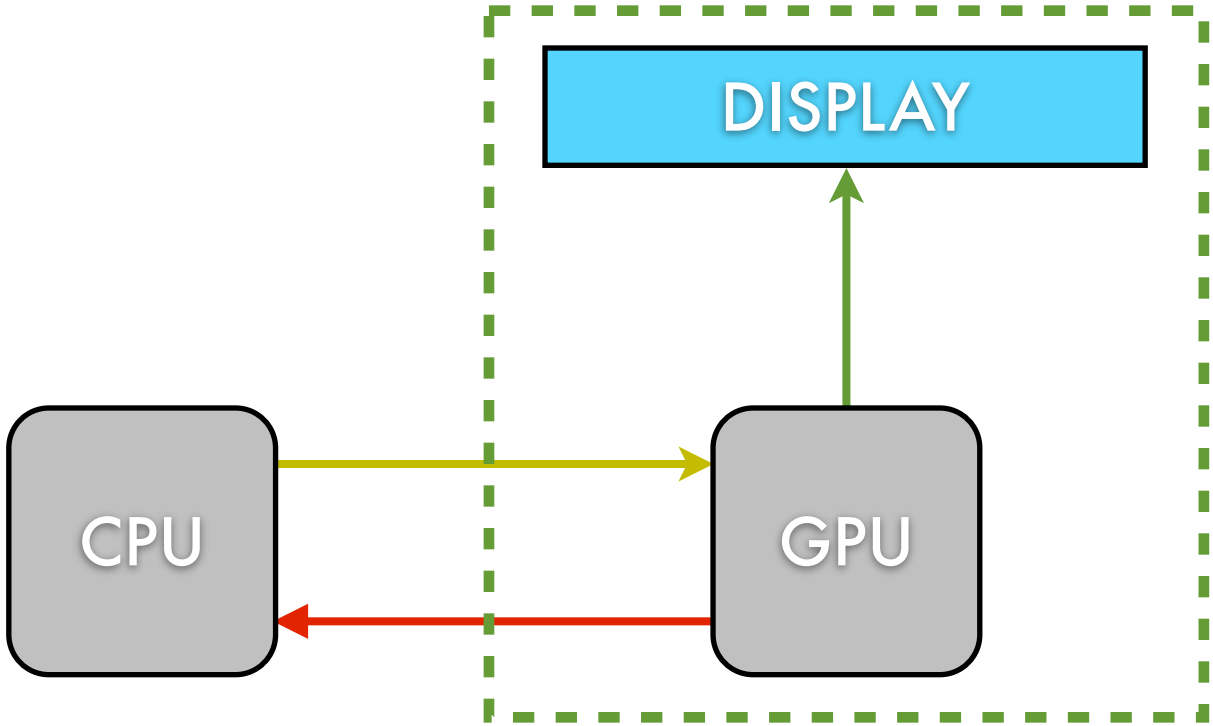
Therefore, any work that can be done on the GPU should be.
Everything else is left to the CPU.

Data Transfer Speeds



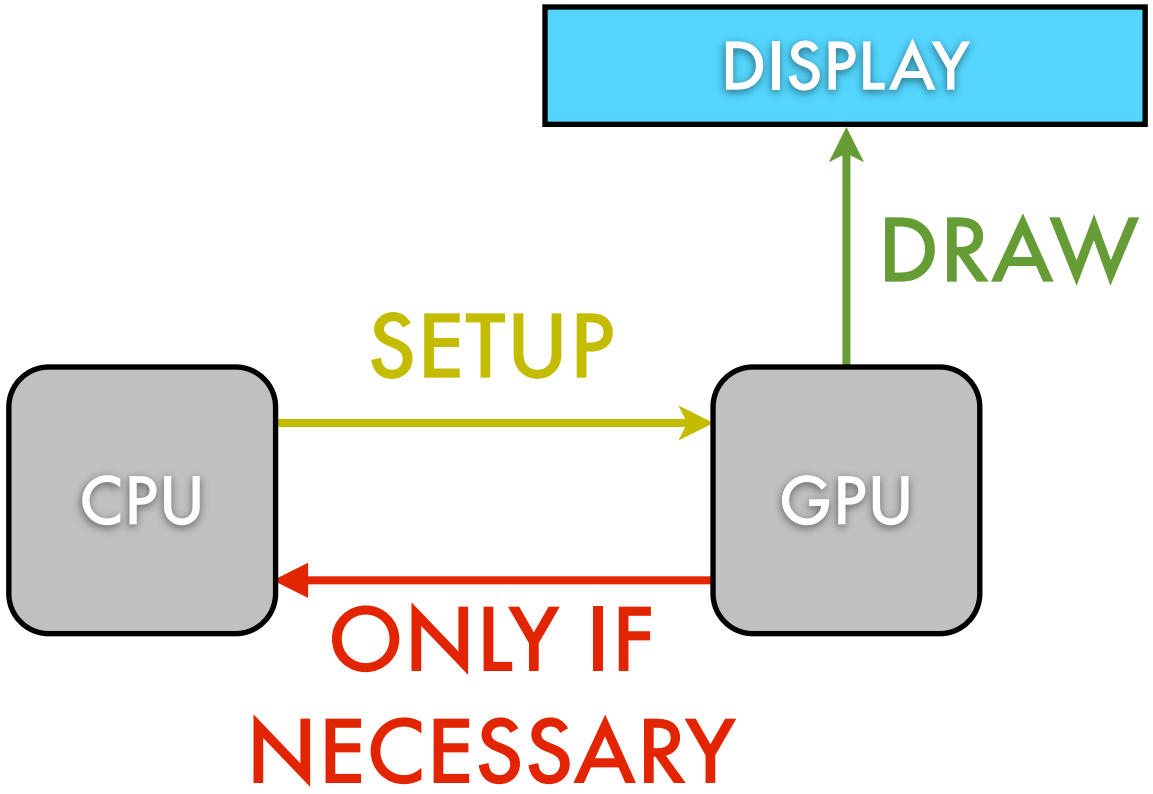
Data transfers between the various hardware components are not equally costly.

The Implications of Data Transfer Speeds



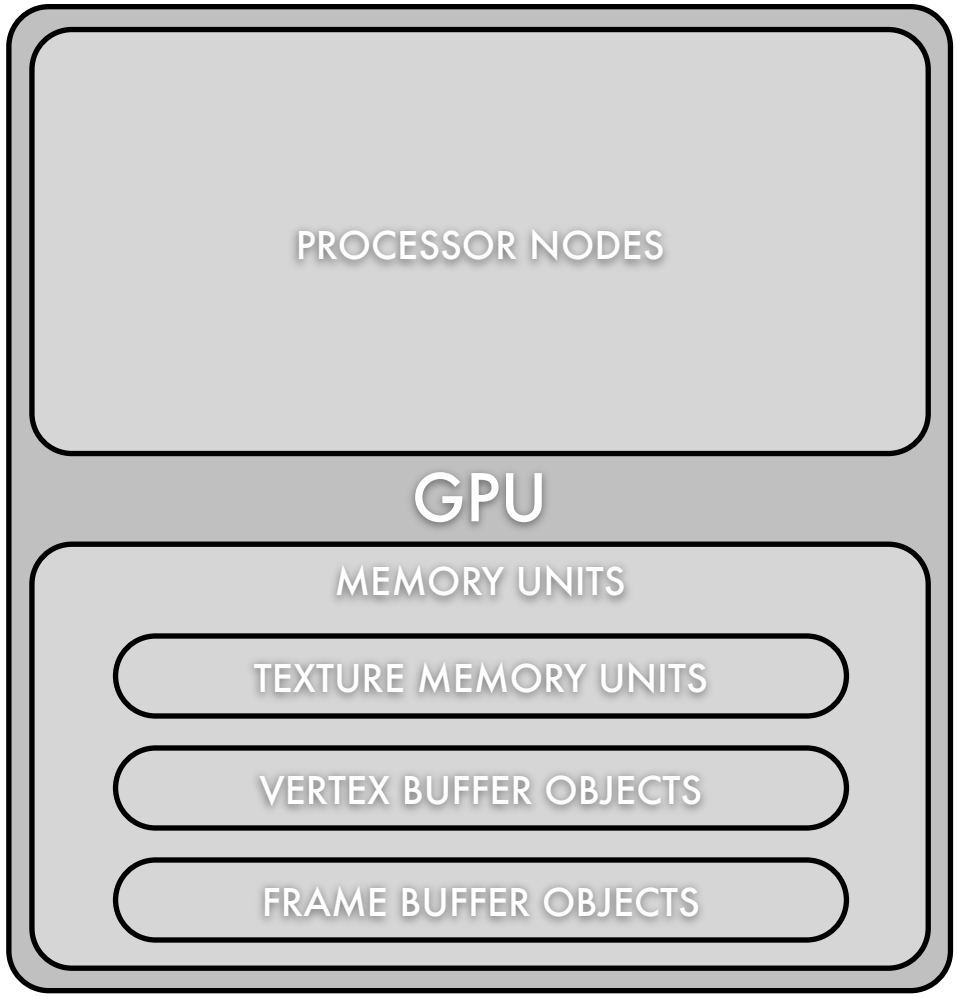
We should design software that keeps
the majority of transfers between the GPU and DISPLAY.

In other words, it would be best to pass the bulk of our geometric data to the GPU during **setup()**.



Within the **draw()** loop, the GPU should handle most data manipulations with minimal input from the CPU.

This is achieved by pre-loading large data elements such as images and vertex lists into the GPU's persistent memory units.



Graphics APIs and Shader Languages provide tools for accessing and manipulating the data stored in GPU memory with minimal CPU usage.

From main() to Processing

Basic Architectural Patterns in Graphics Software

A pure C, C++ or Java program starts with something like this:

```
int main(int argc, const char * argv[]) {
    for(int i = 0; i < 10; i++) {
        int numsq = i * i;
        printf("%i squared is %i.\n", i, numsq);
    }
    return 0;
}
```

The program enters the main function, executes each command in order and then exits.

This structure is linear and straightforward.
The above program will run to completion in a fraction of a second.

However, we don't want our graphics program to exit in a fraction of a second.

So, we need to expand upon this architecture to make room
for the temporal nature of graphics software.

Processing, oF and Cinder use a “setup-update-draw” design pattern:

```
int counter;

void setup() {
    counter = 0;
}

void draw() {
    int numsq = counter * counter;
    printf("%i squared is %i.\n", counter, numsq);
    counter++;
}
```

This pattern divides software tasks into two basic categories:

- Steps performed once when the application is launched: **setup()**
- Steps performed repeatedly throughout application runtime: **update()** and **draw()**

(Processing consolidates update() and draw() into a single function)

This design pattern is just one of many possible,
but is well-suited to graphics software.

But, main() is still under the hood:

```
int main(int argc, const char * argv[]) {
    // The while-loop below will run forever.
    // We'll eventually need to tie this variable to a keyboard "esc" event.
    bool keepRunning = true;

    setup();
    while( keepRunning == true ) {
        // Processing consolidates update() into draw()
        //update();
        draw();
    }
    // In C/C++, a third category - steps you perform once at the end - can be
    // helpful for proper memory management:
    //shutdown();

    return 0;
}
```

As shown above, the setup-update-draw pattern is created
by adding a single **while()** loop to **main()**.

The program will continue to call **draw()** until the value of **keepRunning** is set to false.
We could also add a delay timer to the loop to ensure that frame drawing is evenly spaced.

We can now pass successive frames to OpenGL.

Building a 3D Graphics Platform

What features should we support in our 3D environment?

Preliminary Feature List for 3D Platform

Geometries

Animation
Mesh
Cylinder
Cone
Torus
Sphere
etc.

Cameras

Animation
Orthographic
Perspective

Lights

Animation
Ambient
Directional
Point
Spot

How should we organize the contents of this environment?

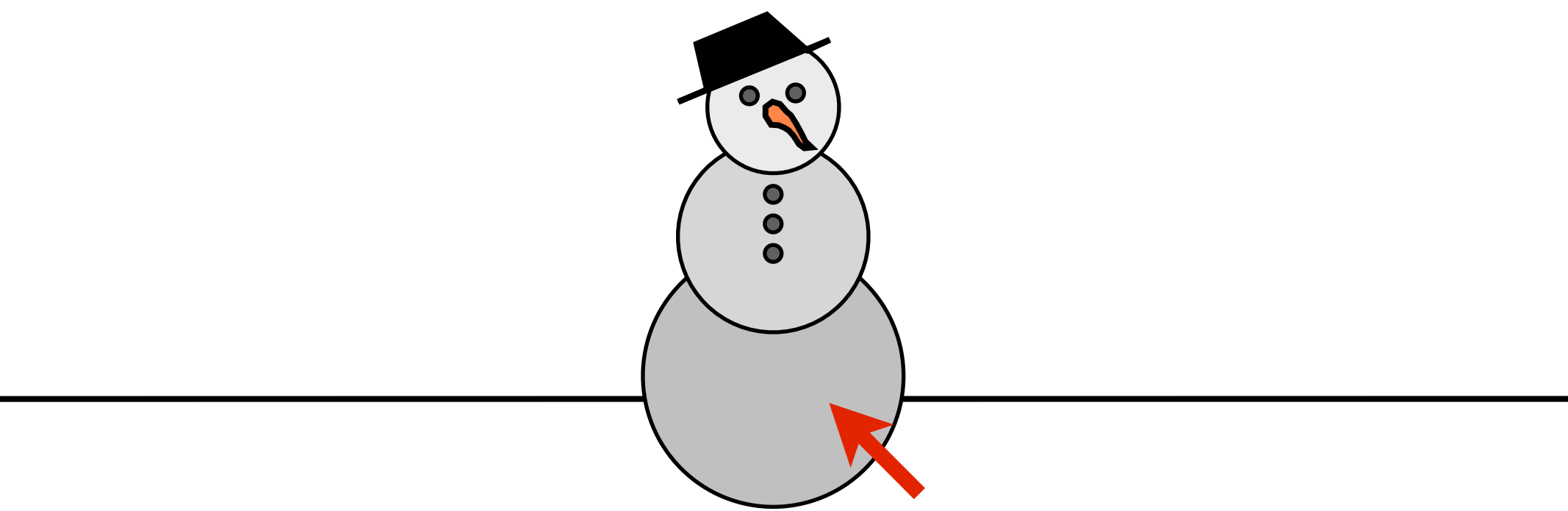
This organization is simple and straightforward:

```
ArrayList mObjects;

void setup() {
    mObjects = new ArrayList();
    mObjects.add( new SphereObject( 0.0, 0.0, 0.0, 50.0 ) );
}

void draw() {
    int tCount = mObjects.size();
    for(int i = 0; i < tCount; i++) {
        mObjects[i].draw();
    }
}
```

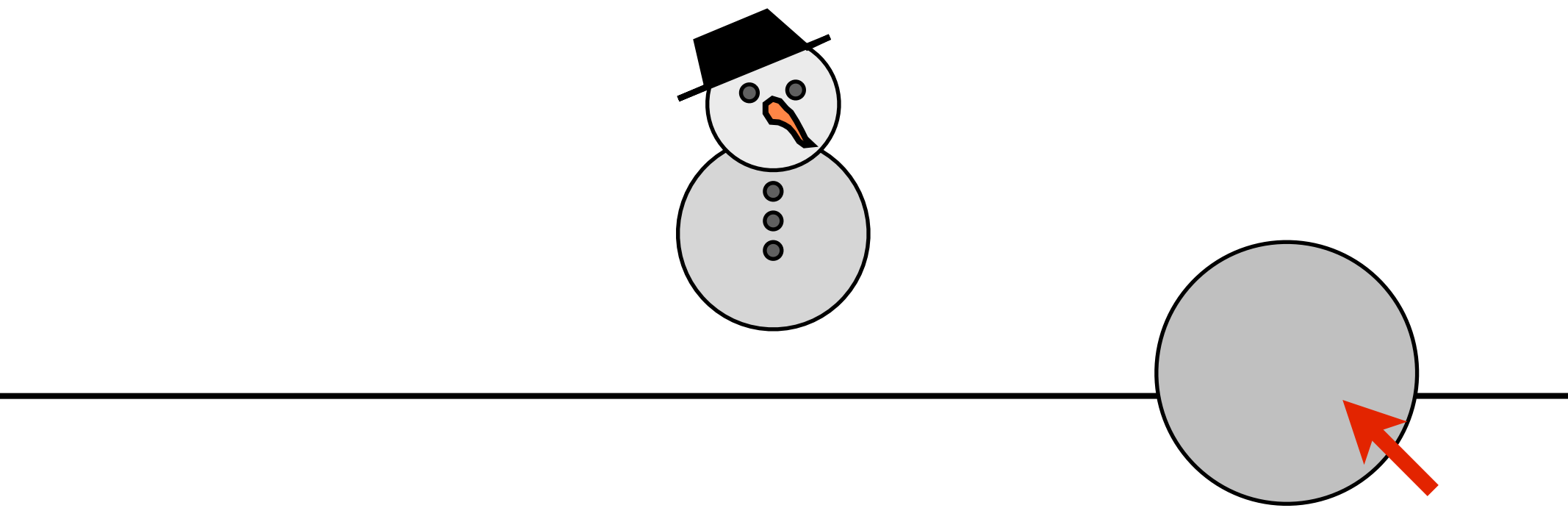
The basic idea here is that we would add a bunch of objects to an ArrayList during **setup()** and then simply draw each object every frame.

This organization is sufficient for some purposes, but...

What if we want to move this snowman?



We probably don't want this:

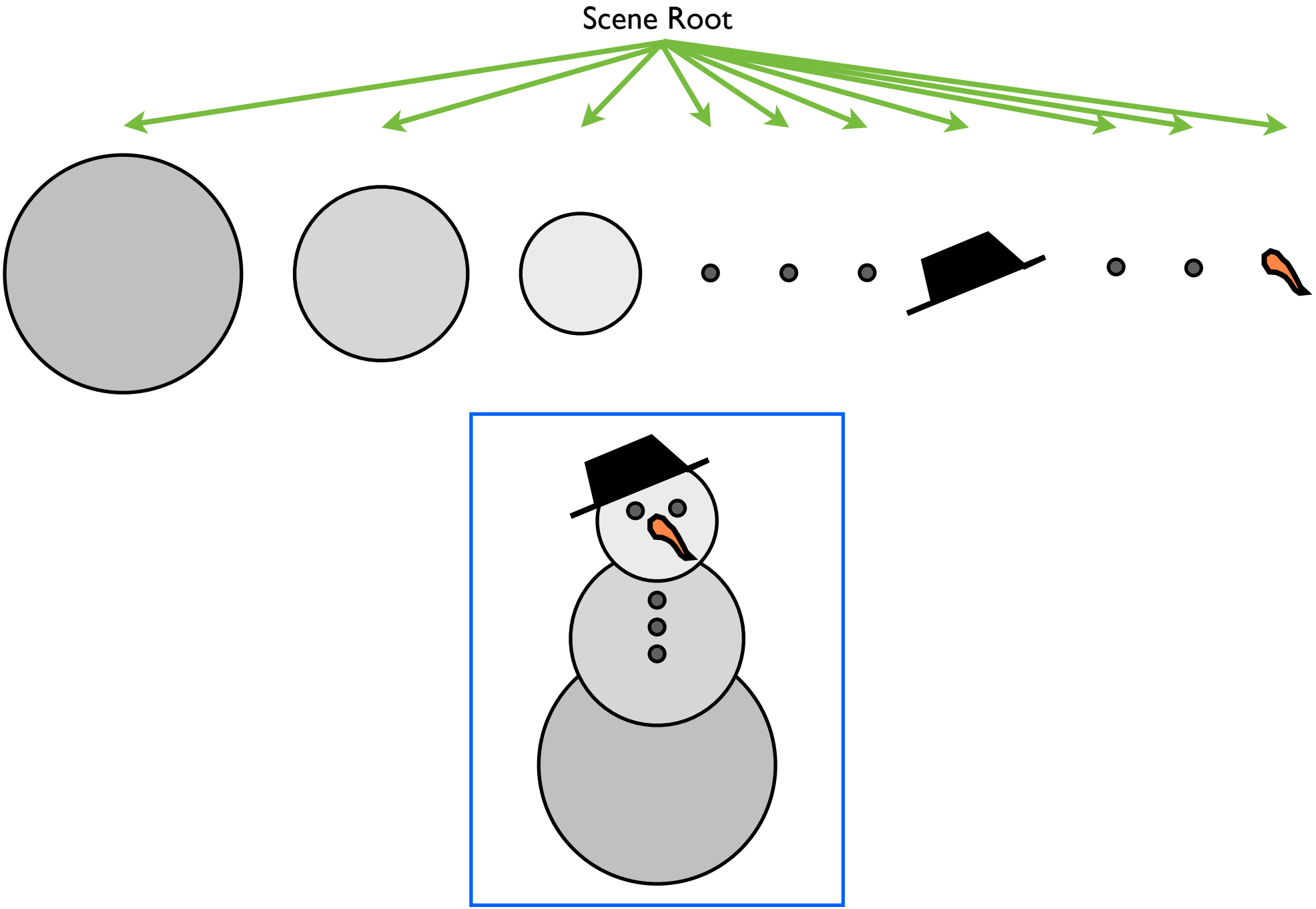


We want this:

In the first interaction, the parts of the snowman were stored non-hierarchically in a single ArrayList.

To move the entire snowman, we would need to move each component individually.

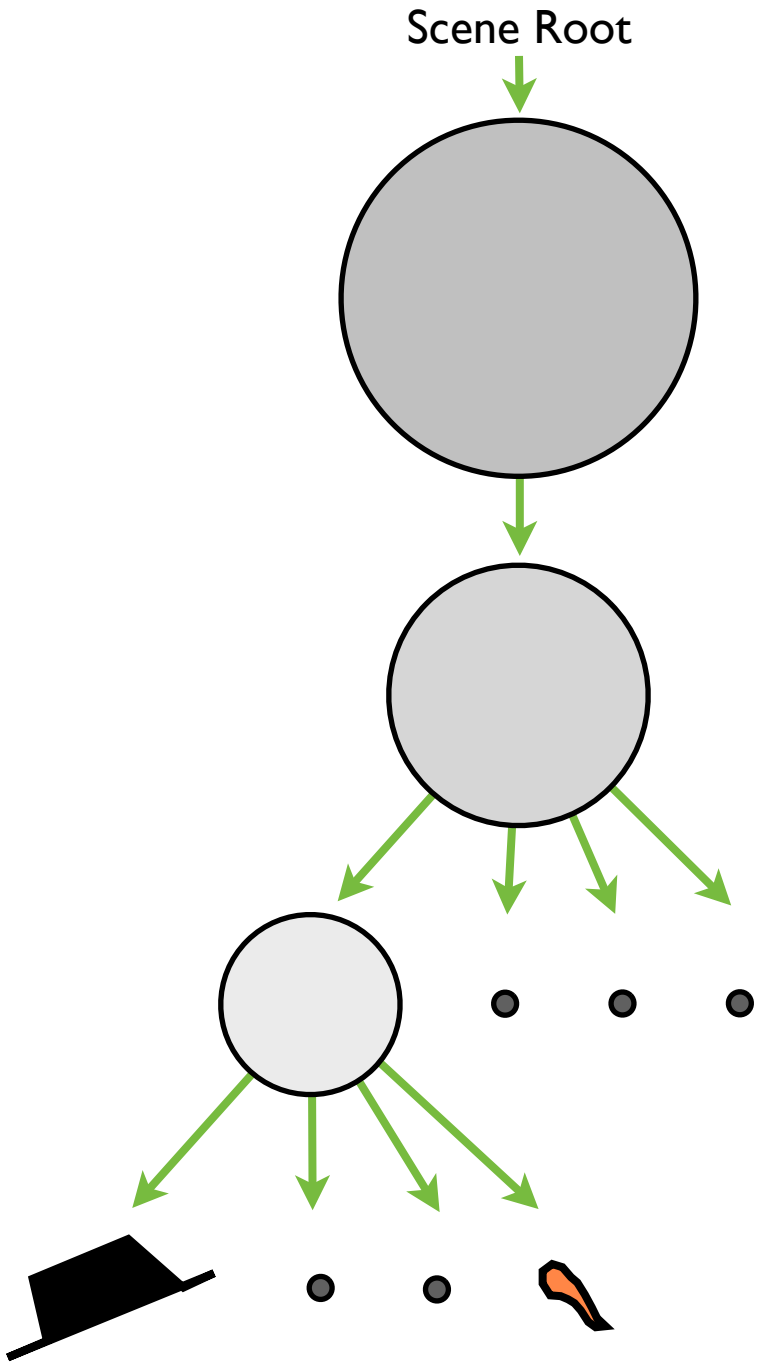
Non-Hierarchical Scene Structure



In the second interaction, the snowman moved as a single object when we grabbed it by its base component.

To achieve this, we used a hierarchical organization of component objects, called a *scenegraph*.

Scenegraph Hierarchy



A Basic Scenegraph Node

```
class NodeBase {

    String                mName;
    NodeBase              mParent;
    ArrayList<NodeBase> mChildren;

    boolean               mVisibility;

    // ...
}
```

Goto Section Examples: AoGP_SE_001