Lecture 24
Performance Tricks

Warning...

Graphics performance is a moving target

Focus on:

- methods that are generally useful (for other reasons)
- methods that are really important in 2022
- methods that are useful for your project

- methods that help us understand the pipeline

Warning (2) (for class...)

"Hardware" is (relatively) fast

- once things get to the "graphics driver"
- you can process lots of triangles/vertices/fragments very fast

  If it is big "blocks"

Avoid obvious problems

- lots of computations in JavaScript (e.g., per vertex)
- redundancy (extra copies of things)
- re-creating objects often

More relatistically

"Real" systems (games, etc.) - pre-organize data (to encourage this)

Engines that encourage doign things efficiently

Lots of initial setup

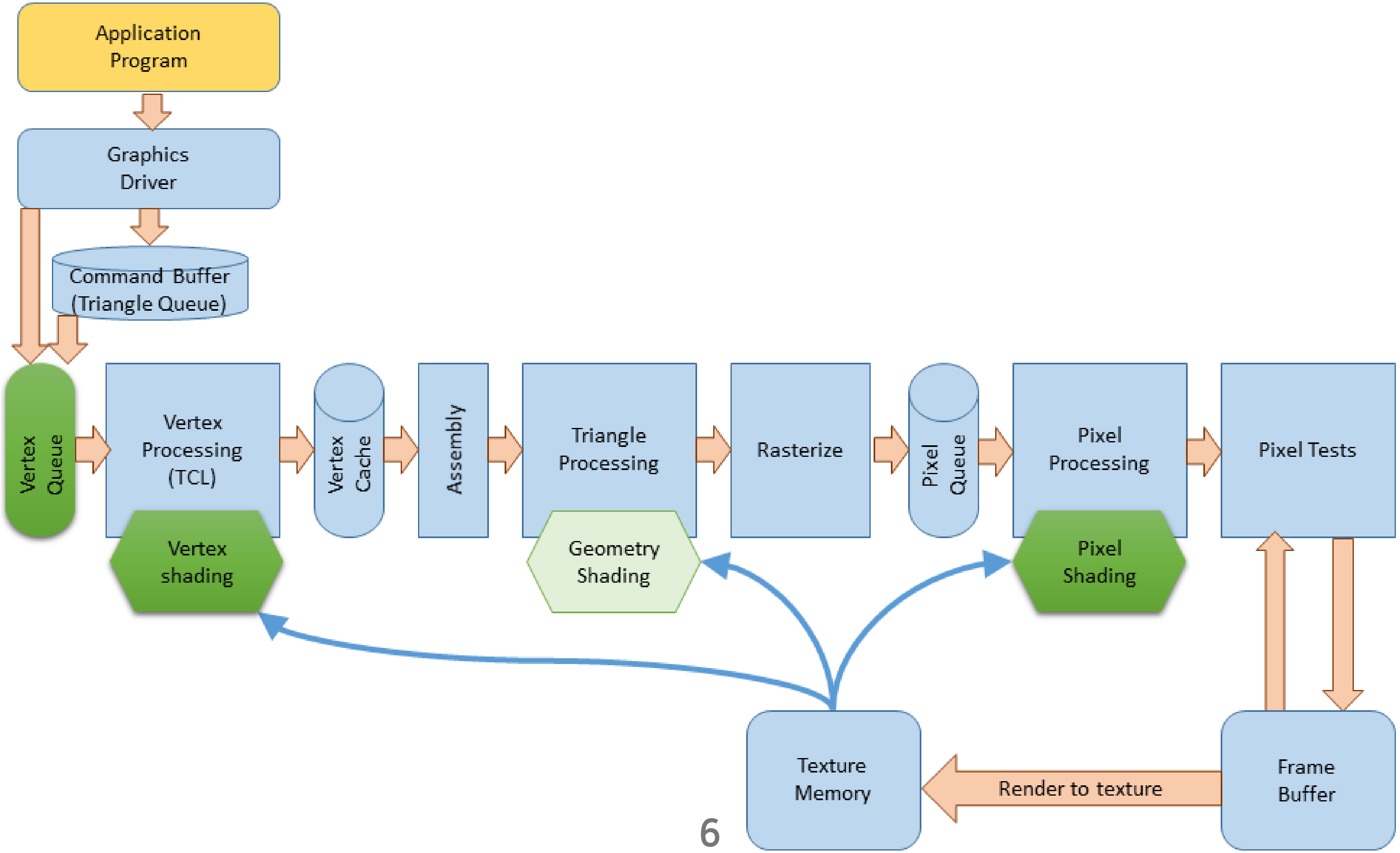
Less work "per-frame"

Amortization - preparation work pays off since it is used over and over

Where are the bottlenecks?

- getting objects and texture to the hardware
  - use large objects and textures
  - animate by deformation
- getting complex lighting
  - use environment map tricks
- too much time shading (fragments)
  - avoid shading what we can't see

The Pipeline - where can things go wrong?



How much work do we do?

- Process the triangle
- Rasterize
- Process Fragments
    - multiple texture lookups
    - texturing computations
    - shadow maps, environment maps, ...
    - potentially many lights, ...

And then...

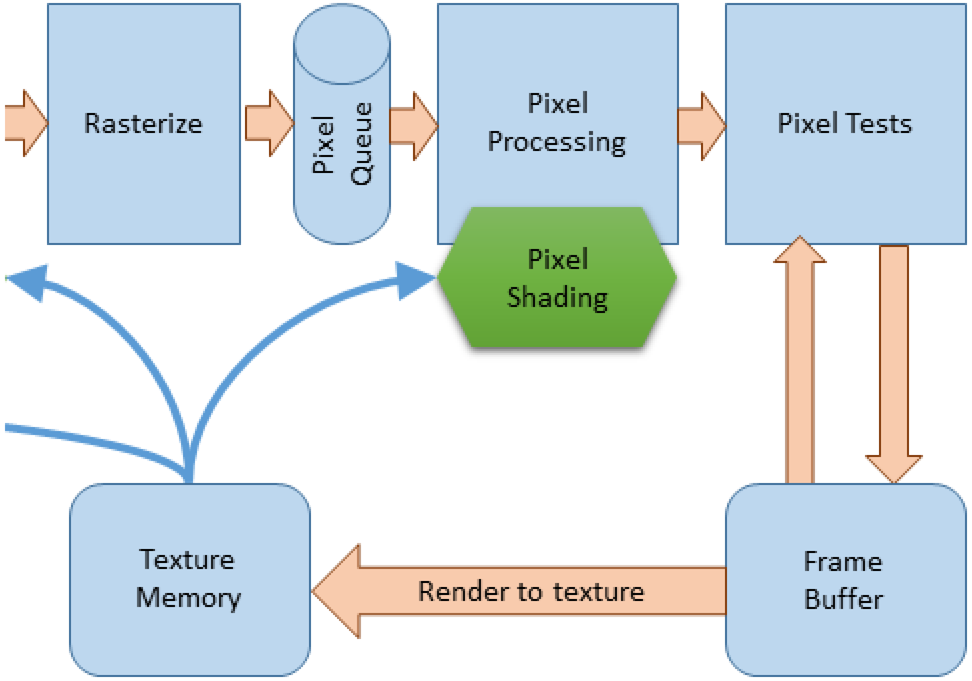
maybe it fails the Z-test (or something over-writes it)

Idea 1: Early Z

If the fragment shader doesn't change the Z coordinate...

Do the Z-Test **before** pixel processing

- this seperates pixel read from pixel write
- can create timing issues



Idea 2: Two Passes

1. Draw with a really simple shader (just write Z)
2. Draw with the early Z-test (do Z equals)
    - in theory, only one fragment passes the test

Good:

- shade each pixel once
- no sorting required

Bad:

- two passes
- still need early Z cutoff (but its simple - since only 1 accept per pixel)

Idea 3: Defered Rendering

This is a big deal in modern games

This isn't something you do yourself in THREE

Deferred Shading/Rendering

1. Don't shade the fragments when we draw them
2. Store information needed for shading in a **Geometry Buffer**
3. Make a second pass where colors are computed
    - possible multiple passes
    - each pixel is considered once
    - one big polygon that covers screen

A Note on Terminology

Deferred **shading**

Deferred **rendering**

Deferred **lighting**

Different sources use these terms differently

The basic ideas are the same

Arguments as to which details go with different terms

Simple Deferred Shading Example

1. Draw all object w/Z-buffer
    - store in G-Buffer
2. Draw one big polygon (screen)
    - shade using G-Buffer

Extensions

1. Loop over light sources (light sums in pixels)
2. Light sources don't need to do entire screen
3. Break things into **tiles**
   - break screen into regions
   - list of triangles for each region (culling)
   - list of lights for each region
   - much smaller memory footprint

Tile-based defered rednering - very common in games

Mobile hardware is explicitly designed for TBDR

Complex Lighting

- How do we have lots of lights?
- How do we have objects act as lights?

Complex Lighting

Simple Lighting: specular vs. diffuse

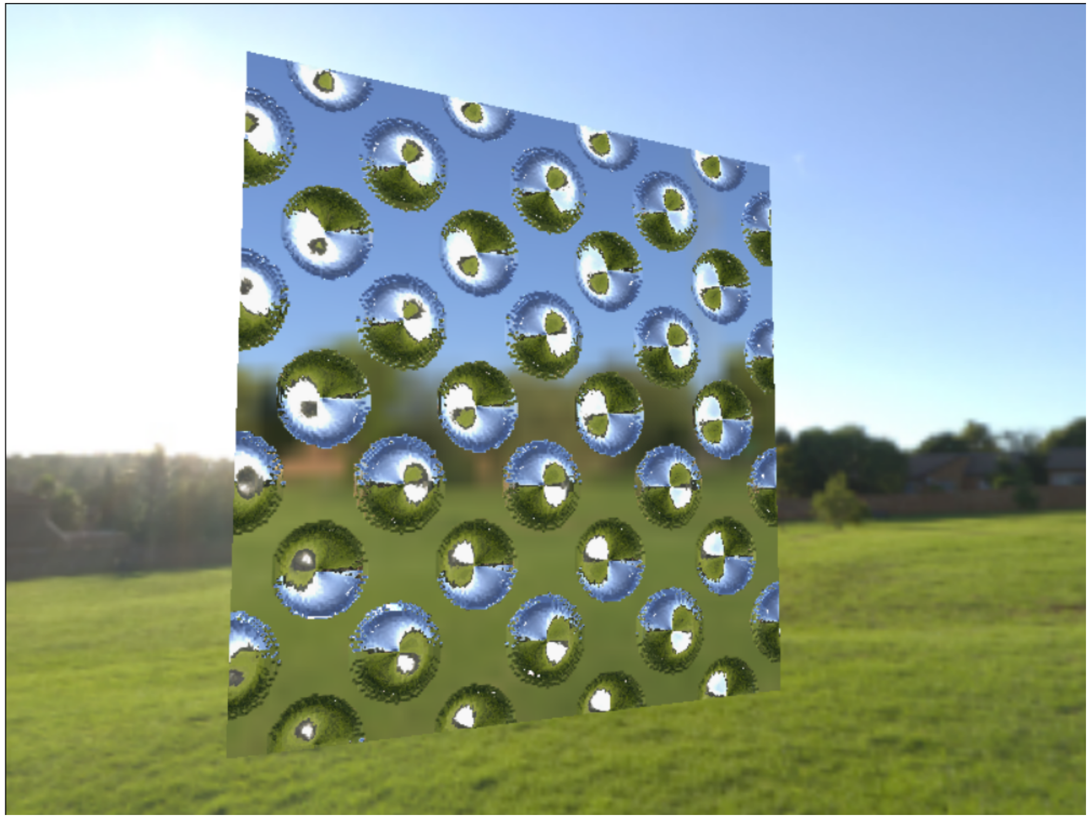
Specular model

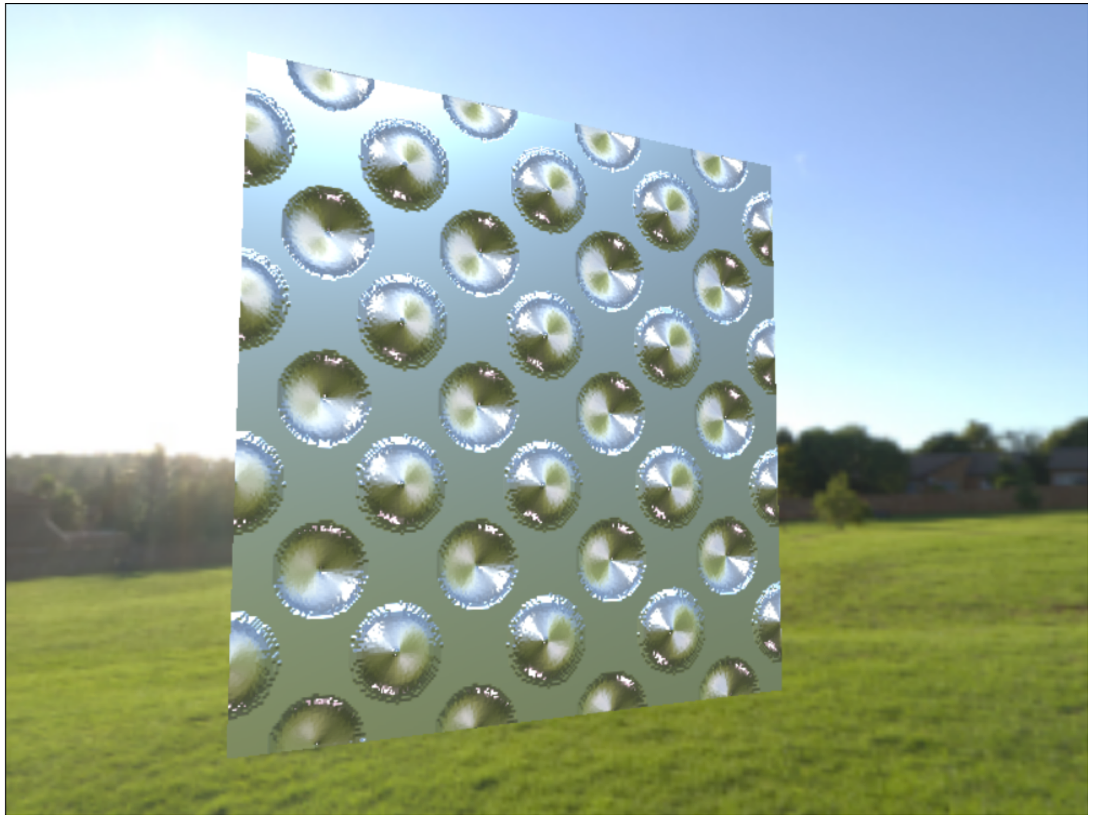
As "shininess" goes to zero, becomes (more like) diffuse

Using with an environment map

As "shininess" goes down (roughness goes up), filter area goes up

Less "reflective"





A Hack way to get fancy lighting

- Capture "main sources" in environment map
- Independent of number of lights
- Independent of complexity of lights
- Potentially Dynamic (with Dynamic Environment Map)
- Potentially capture real lighting
    - used for visual effects in movies

Something more practical (for your project)

The biggest problem students run into:

Too Many Textures

Do We Need So Many Textures

Textures are expensive!

- need to load them
- need to process them
- need to store them
- need to set them up
- need to read from them (caching)

Tricks to make it work better

- reuse (load the texture once)
- put many textures in one image
    - Texture Atlas

Another Practical tip...

Graphics **commands** are expensive

Drawing an "object" requires setup

All the triangles in a group get processed quickly

Lots of small objects... lots of time switching

Animation in THREE

- it's a scene graph
- we update the objects
- we ask three to redrawn the world
- **other systems are similar**

Warning:

- not everything is easy to change
- hard to understand unless we know what is happening inside

What is easy to animate?

Easy

Change a transformation

Change a material property (*)

Change a light property

Properties designed to be animated

- small number of numbers
- parameters not buffers

Hard

Change points in a Mesh

Change a material or texture

Change a light type

- Anything that requires sending large data to the **hardware**
- Anything that requites recompiling a **shader**

One big object?

One group of triangles

Send the data once

Low-level (hardware?) loops over lots of triangles

But...

How do we animate it?

What if we want to transform parts differently?

Animation by Deformation

How to change shape - without changing large numbers of vertices

Why?

1. performance (simple computations per-vertex)
    - no need to send mesh to graphics hardware each frame
    - per-vertex computation with limited data
2. authoring (artists don't have to sculpt every vertex)
    - design base shape and make coarse adjustments
3. storage (don't need to remember every vertex in every pose)
4. re-use (apply deformations to different base shapes)

Strategies

1. Basic Transformations
    - mutliple transformations
    - skinning / skeletons
2. Complex Transformations (Deformations)
    - bends/twists
    - Free-form deformations
    - Cages
3. Multiple Meshes (morphing)

Animation by Transformation

Translate or rotate...

1. Change each vertex
    - compute N vertices
    - transmit N vertices between CPU and GPU
2. Change a transformation
    - change 1 number (maybe 12 for a matrix)
    - send 1 matrix to GPU

Downside: limited things we can do (with simple transformations)

Matrix Palette

1. Pass multiple matrices (an array of them)
2. Each vertex specifies which matrix it is part of
    - attribute

Specifying which matrix

1. Give the number of the matrix
2. Give a vector of weights
   - allows for blending

This is really a setup for skinning...

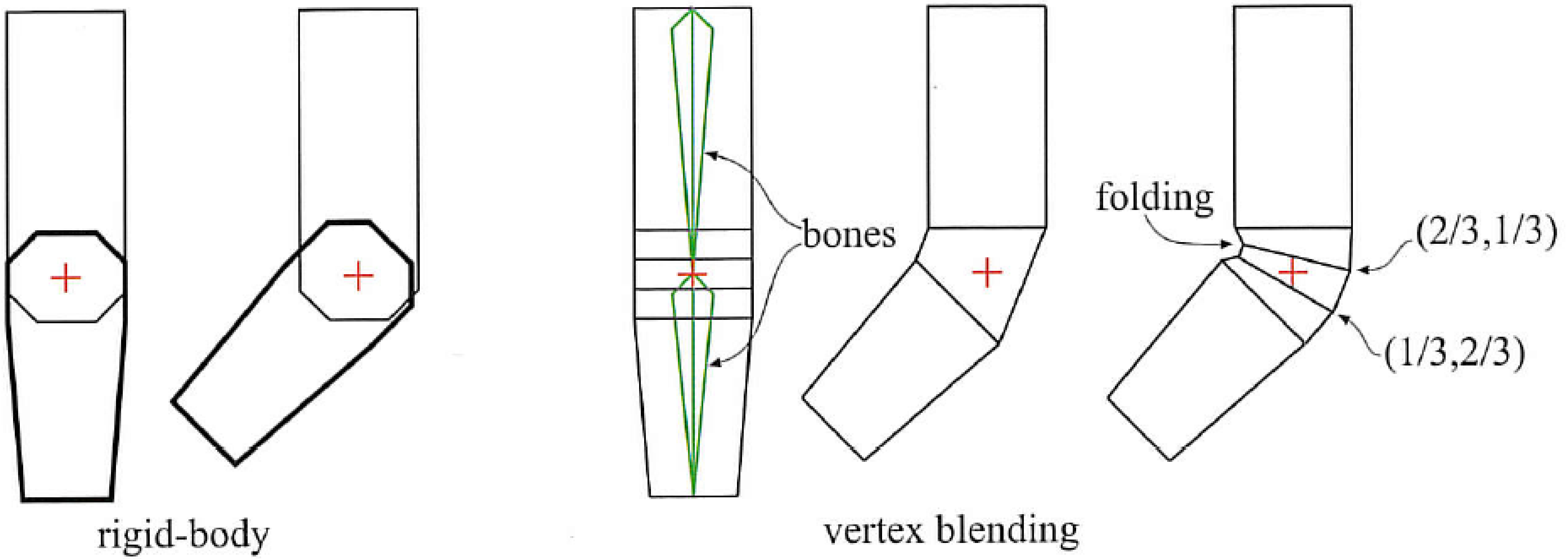
Skinning

1. Each point in one transformation
2. A point in multiple transformations
    - the point has a different "initial" position
    - the transformations are relative

Skinning

Linear Blend Skinning

- Each vertex in multiple coordinate systems (weighting)


bones
folding
(2/3,1/3)
(1/3,2/3)
rigid-body
vertex blending

Blend Matrices?

$$p_s = \alpha M_1 p_0 + \beta M_2 p_0$$

is the same as

$$p_s = (\alpha M_0 + \beta M_2) p_0$$

If M_1 and M_2 are rotations, scaling/adding doesn't give a rotation!

This leads to weird artifacts...

Blend Artifacts

Blend artifacts


Skeletons

Bones as coordinate systems

- how to specify movements?
  - inverse kinematics
  - motion capture
- how to draw the character

Linear Blend Skinning?

- Popular because it is simple
- Artists work around artifacts
- More complex alternatives exist - trade complexity for quality

More Generally...

Make one shape

Deform it to other shapes

- easier to animate
- easier to model/control

Non-Linear Deformations

- Bend/Twist/Other
- Lattice / Free-Form Deformation
- Cages

Deformations as space transformations

$$x', y', z' = f(x, y, z)$$

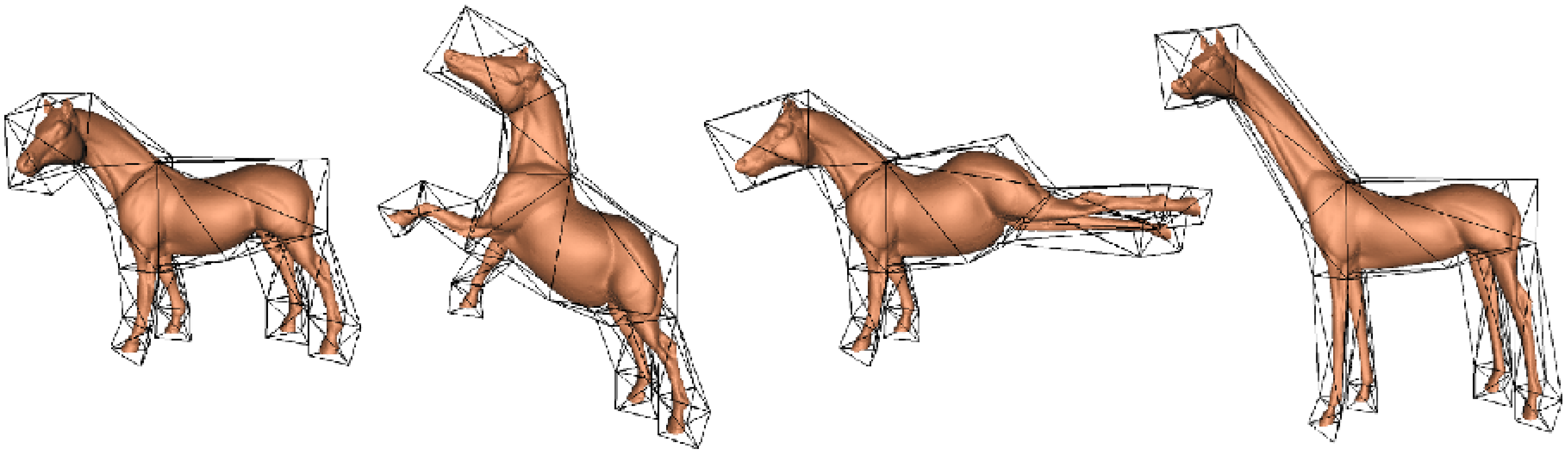
Grid Deformers

Free-Form Deformations (FFD)

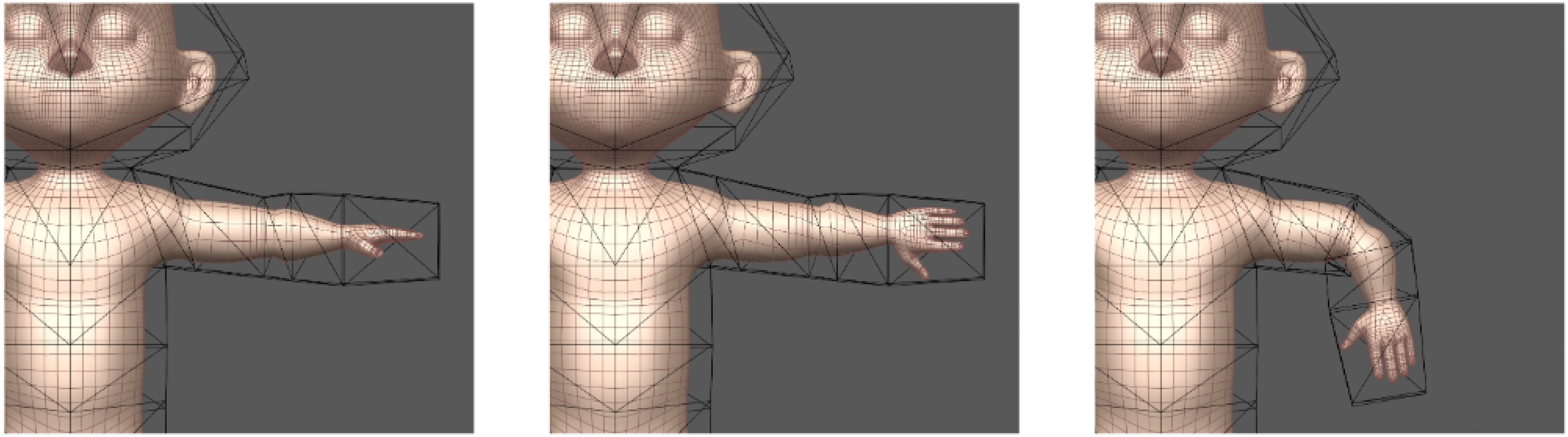
The trick: coordinates in irregular shapes

- Triangles (Barycentric)
- Squares (XY)
- Anything else? (generalized barycentric)

Cages (Coordinate-based Deformations)



Harmonic Coordinates (Pixar 2000s)



https://graphics.pixar.com/library/HarmonicCoordinatesB/paper.pdf

Multiple Mesh concepts

Shape Interpolation (Morphing)

Create multiple copies of the mesh

- each copy is a **morph target**

Vertices interpolate between targets

- blend their positions in each target
- $p = w_1 p_1 + w_2 p_2 + w_3 p_3$ for each vertex

Send all meshes to the hardware

Each frame only changes the weights

Downsides

1. Need to make all the meshes
2. Meshes need to correspond
3. In-between values may not be meaningful
4. Control by blending (not always easy)

In THREE

Built in to THREE!

(see that weird blobby thing in the graphics town demo)

OK, But What Can I Do for my Project?

In Graphics Town you can:

1. Be careful about texture usage (use an Atlas!)
2. Use fewer, bigger objects
3. Use Environment Maps
4. Try Skinning and Morphing (built into THREE)
5. Try implementing complex deformations (good shaders practice)

Avoid doing "stupid" things (things you can't see, redundancy, ...)