Computer Graphics

-Advanced Texture Mapping & Graphics Pipelines-

Oliver Bimber

Course Schedule

Type	Date	Time	Room	Topic	Comment
C1	01.03.2016	13:45-15:15	HS 18	Introduction and Course Overview	Conference
C2	15.03.2016	13:45-15:15	HS 18	Transformations and Projections	Easter Break
C3	05.04.2016	13:45-15:15	HS 18	Raster Algorithms and Depth Handling	
C4	12.04.2016	13:45-15:15	HS 18	Local Shading and Illumination	
C5	19.04.2016	13:45-15:15	HS 18	Texture Mapping Basics	
C6	26.4.2016	13:45-15:15	HS 18	Advanced Texture Mapping & Graphics Pipelines	
C7	03.05.2016	13:45-15:15	HS 18	Intermediate Exam	
C8	09.05.2016	17:15-18:45	HS 18	Global Illumination I: Raytracing	
C9	10.05.2016	13:45-15:15	HS 18	Global Illumination II: Radiosity	Conference / Holiday
C10	31.05.2016	13:45-15:15	HS 18	Volume Rendering	
C11	07.06.2016	13:45-15:15	HS 18	Scientific Data Visualization	
C12	14.06.2016	13:45-15:15	HS 18	Curves and Surfaces	
C13	21.06.2016	13:45-15:15	HS 18	Basics of Animation	
C14	28.06.2016	13:45-15:15	HS 18	Final Exam	
C15	04.10.2016	13:45-15:15	TBA	Retry Exam	

NEXT ICG LAB TALK:
26. APRIL 2016, 4:00PM

Institute of Computer Graphics

Prof. Daniel Weiskopf

University of Stuttgart

Eye Tracking and Visualization

Computer Science Building (S3)

Room S3 048

JKU

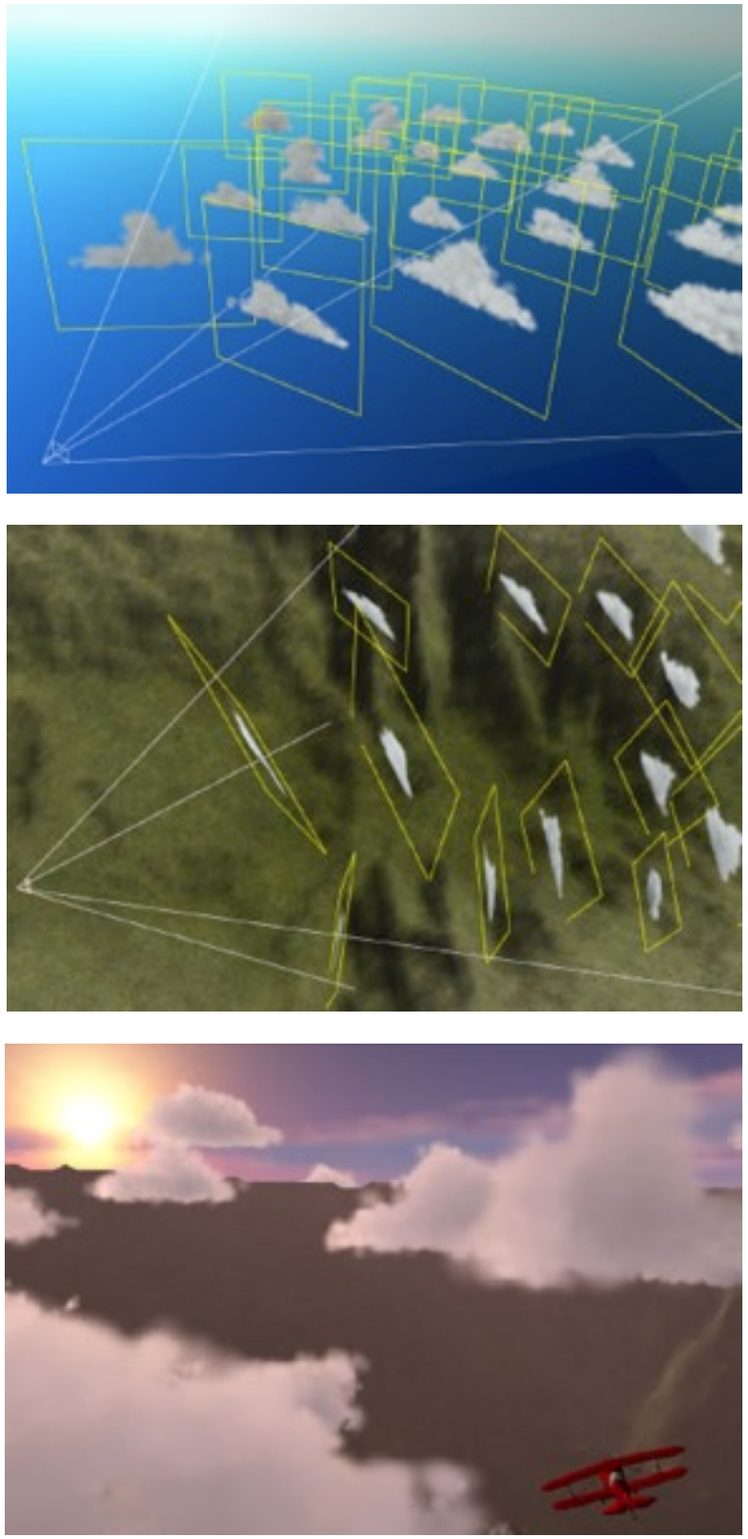
For more information about our talks visit
http://www.cg.jku.at/talks/invited

Texture Mapping Continued

How much is Texture-Mapped?

Billboarding

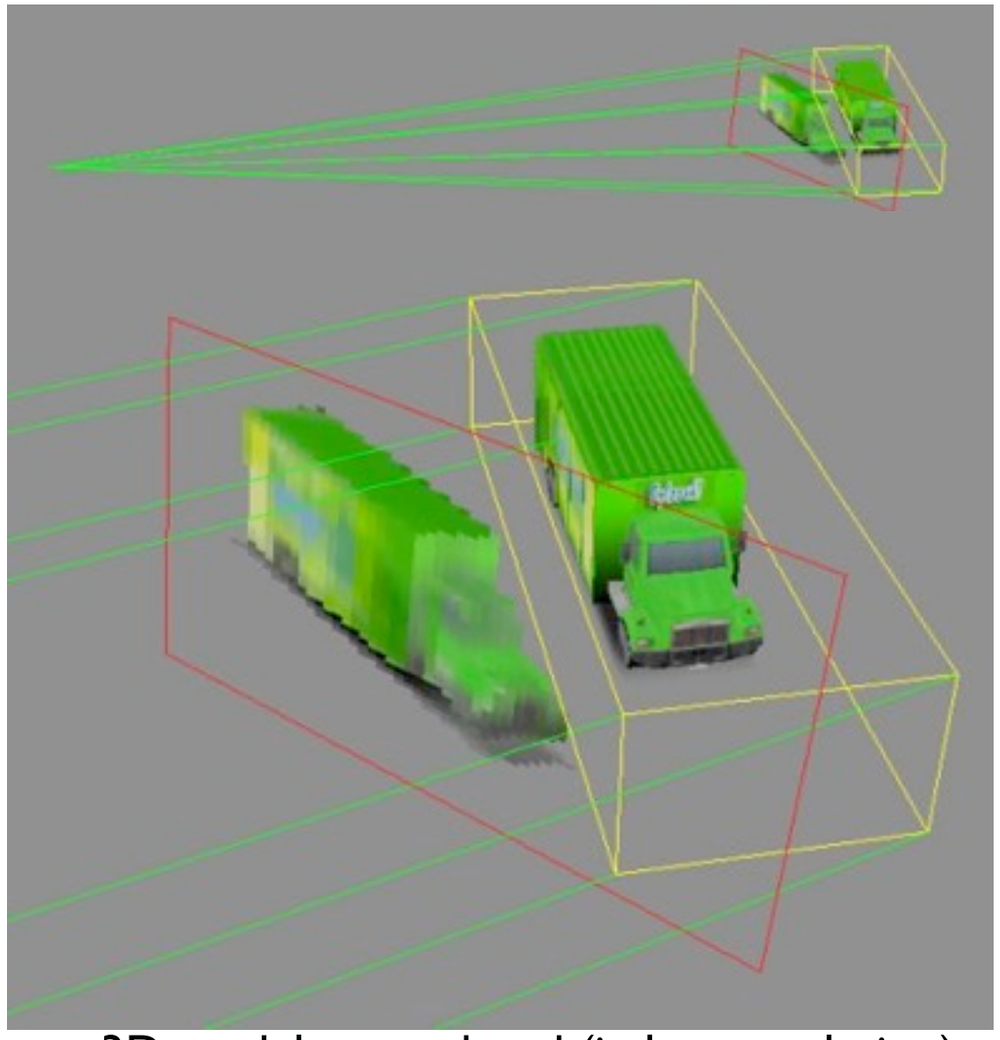
- A textured polygon that faces the camera (usually perpendicular to viewer direction) is called billboard
- Billboarding combines alpha texturing with animations
- The (relatively simple) billboard geometry is transformed in 3D while its texture is updated
- The texture can contain pre-rendered images of more complex 3D scenes



billboarding for cloud rendering

Imposters

- The bilboard‘s texture has to be replaced from time to time to aviode extreme distortions
- Every time the texture is re-generated, the corresponding object has to be rendered and we loose the gain in performance of IBR
- Thus, we want to re-use a pre-rendered texture as often as possible
- But what is a good balance between aceptable distortion and performance?
- Bilboards that re-render its textures depending on the estimated distortion from time to time are called imposters
- Different level-of-details can also be considered (i.e., the impostor texture does not need a higher resolution than the resolution on screen area it is projected to)

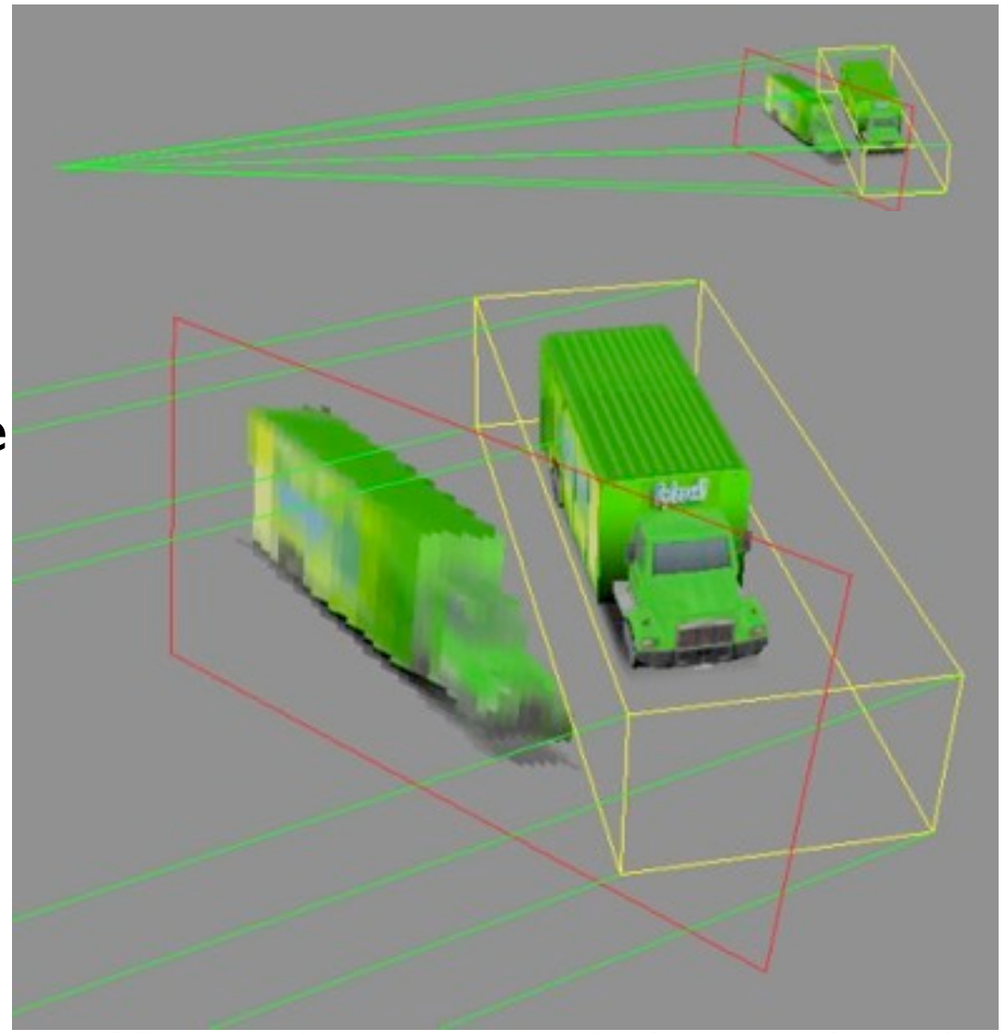


3D model re-rendered (in low resolution) to update an impostor for new perspective

3D model next to its 2D impostor

Imposters

- To re-render an impostor, the virtual camera points at the center of the corresponding object‘s bounding box (this corresponds also to the projected center of the impostor)
- The bounding box (or other bounding volume, such as sphere) usually defines the camera‘s frustum
- After the impostor‘s texture is rendered, image processing can be optionally applied (like MIP-mapping, smoothening, etc)
- In general, it is desired to re-use the same impostor texture over several frames
- This works only, if object and viewpoint did not change much relative to each other (called frame-to-frame coherence)
- If distance between camera and object is large, perspective distortion is minimized and frame-to-frame coherence maximized
- Thus, impostors are useful for rendering distant objects rapidly

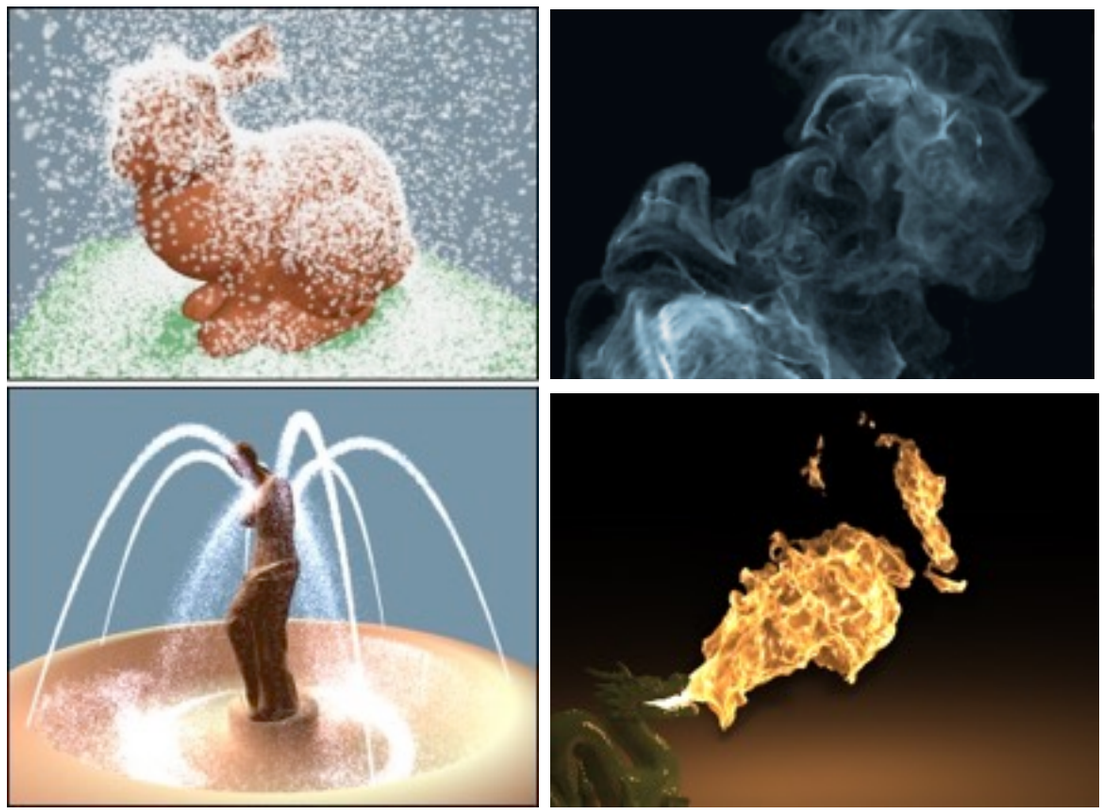


3D model re-rendered (in low resolution) to update an impostor for new perspective

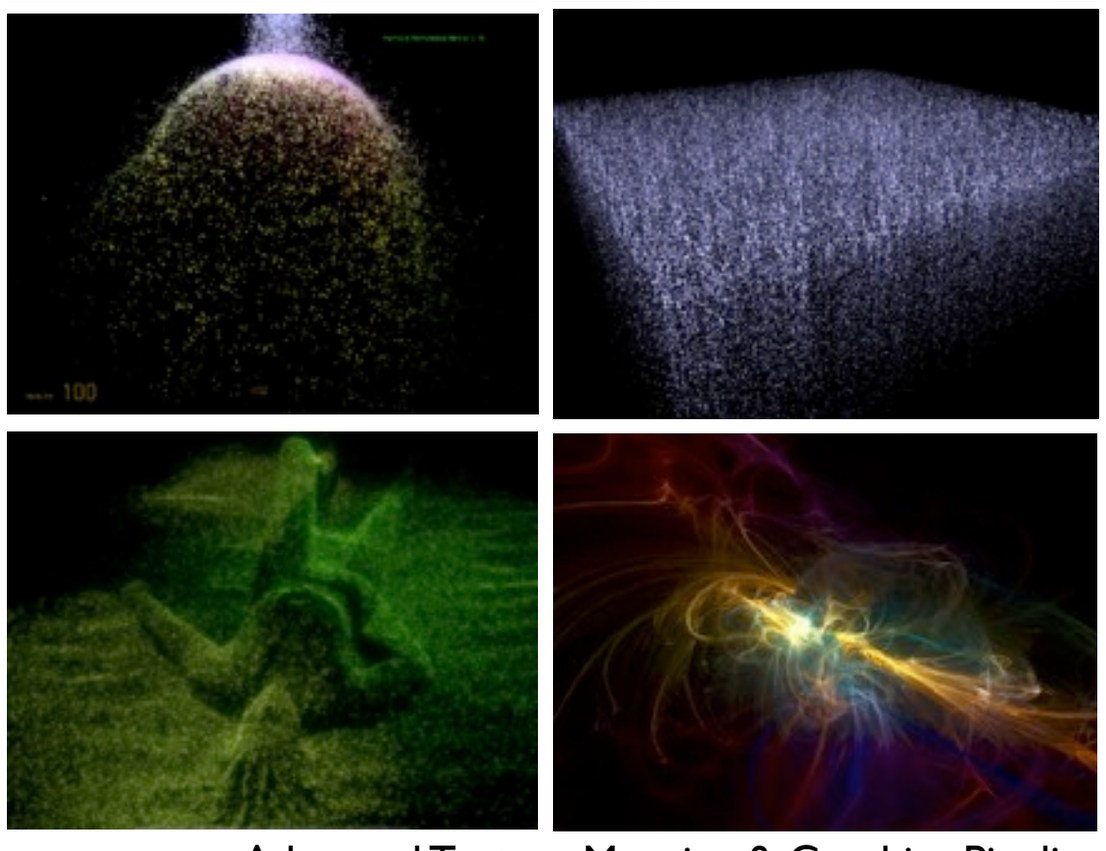
3D model next to its 2D impostor

Particle Systems

- A particle system is a set of small objects (particles) that are animated using an (e.g., physics-based) algorithm
- Examples are smoke, fire, water flows, explosions, etc.
- Bilbords can be used for representing particles



- If particles are round, orientation does not matter
- In many cases, even simpler representations are used, such as lines and points (e.g., point-based rendering, splatting)



Particle Systems

- Hard particle: regular flat billboards
  - seam between particle and other objects
- Soft particle: fragment shader
  - sample from the depth buffer (GPU support)
    - fading the particle if get closer to other geometry
- Advanced types: volumetric shadowed particles (shadowing on particles), interactive dynamic particles (e.g. interactive smoke particles with fluid dynamics)

hard- vs. soft-particles

Glare Effects

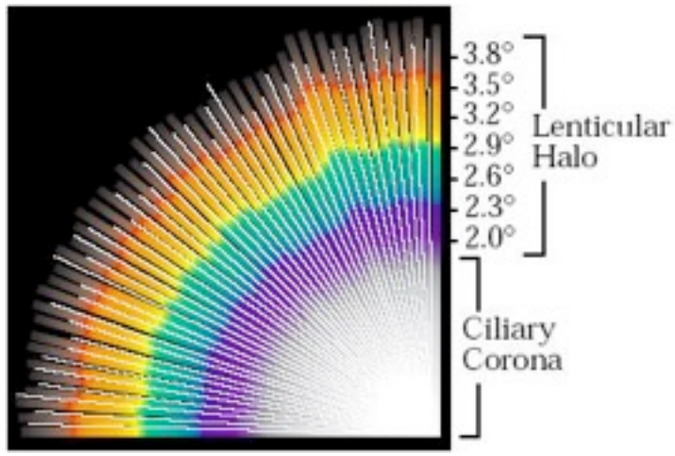
- Lens flare is caused by the lens of the eye or a camera when directed at bright light

- It consists of a halo (lens material refracts light with respect to its wavelength) and ciliary corona (comes from the density fluctuation in a lens, and appears as rays radiating from a point, wich may extend the halo)

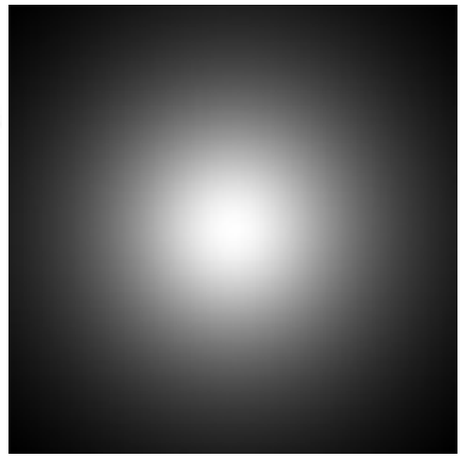
- Bloom is caused by scattering of light in the lens or other parts of the eye or on the sensor camera sensor / film

- Lens flare and bloom are called glare effects

- These glare effects are rendered with impostors or bilboards (multiple textures on different bilboard planes)



flare



bloom

flare and bloom in graphics

flare and bloom in reality

Full-Screen Billboarding and Skyboxes

- A screen aligned billboard that covers the entire view is called full-screen billboard
- They can be used to change the appearance of the scene (e.g., day/night view)
- If an environment map is actually rendered centered to the viewpoint (not only being used for creating reflections on objects), then this is called a skybox (in case cube maps are used)
- Skyboxes can be used for rendering far away objects (like stars) that will not change (much) if the viewer changes
- If the application enforces static viewpoint positions (not orientations) then the entire scene can be rendered with skyboxes (if static) - see QuickTime VR

example for skyboxes

Recap: Environment Maps

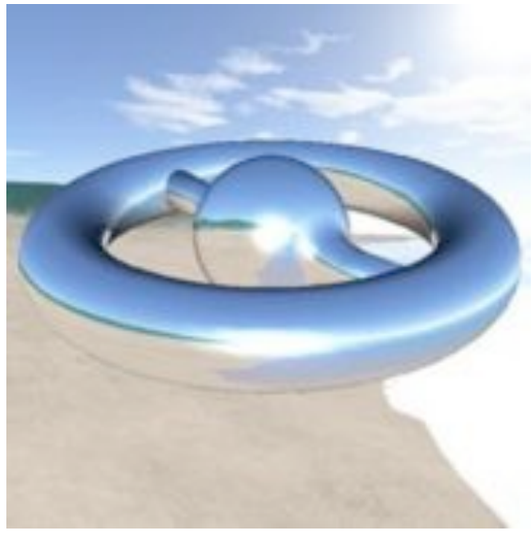
- Obviously, the texture mapping concept is very powerful and supports much more than simple wall-papering

- Environment maps are yet another example - they allow simulating specular reflections of backgrounds (not only of light sources, as we have seen earlier) on the surface

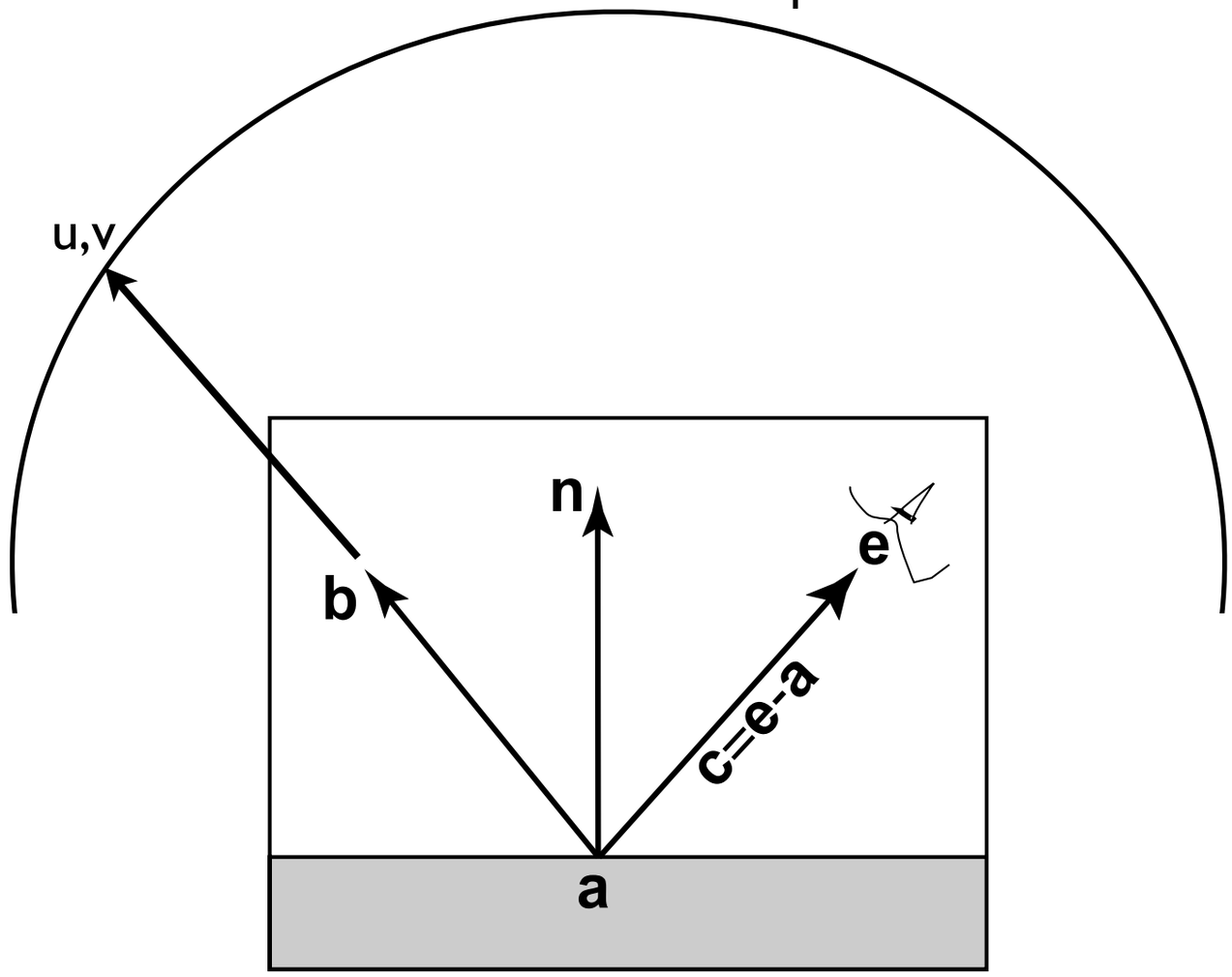
- One can think of a non-planar, parameterizable texture surface that surrounds the scene

- The texture coordinates that are applied to a surface point (a) depend on the reflection vector (b) on the surface

- There are different ways to store environment maps: eg. spherical index table or cube-based table

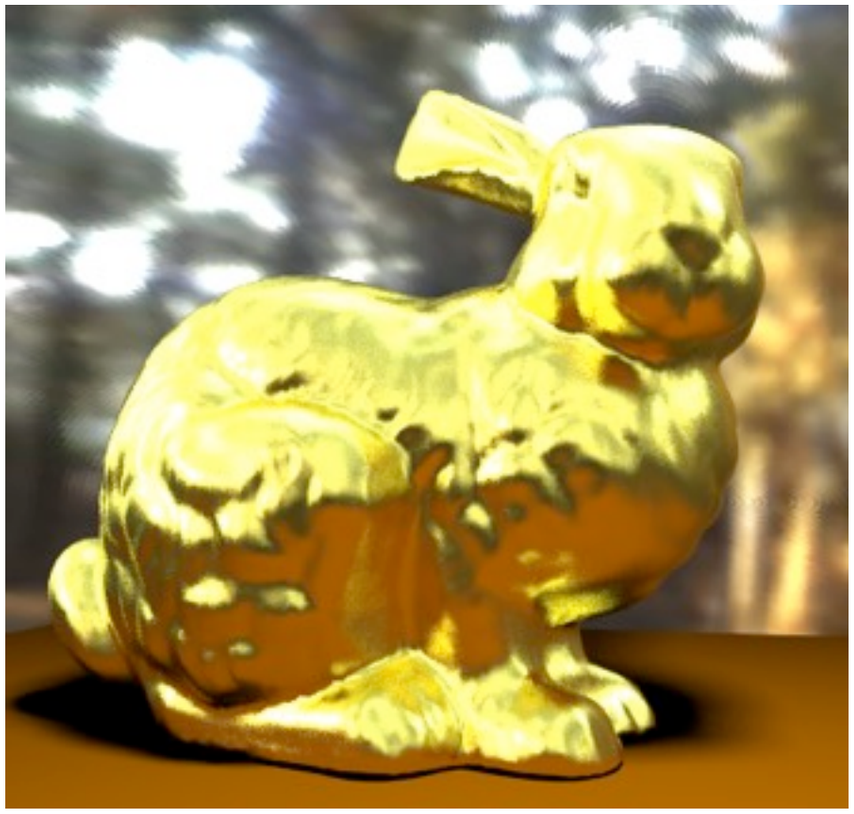


environment map



More Advanced Environment Mapping

- How can we use environment mapping techniques to render (in real-time) glossy (but not perfectly specular) materials?
  - environment mapping / cube mapping only supports specular reflections
  - raytracing is too slow



More Advanced Environment Mapping

- How can we use environment mapping techniques to render (in real-time) glossy (but not perfectly specular) materials?
  - environment mapping / cube mapping only supports specular reflections
  - raytracing is too slow
- How can we apply more complex environment lighting?
  - ambient term in local shading is a poor approximation (no directional information)
  - radiosity is too slow

Environment Mapping Filtering

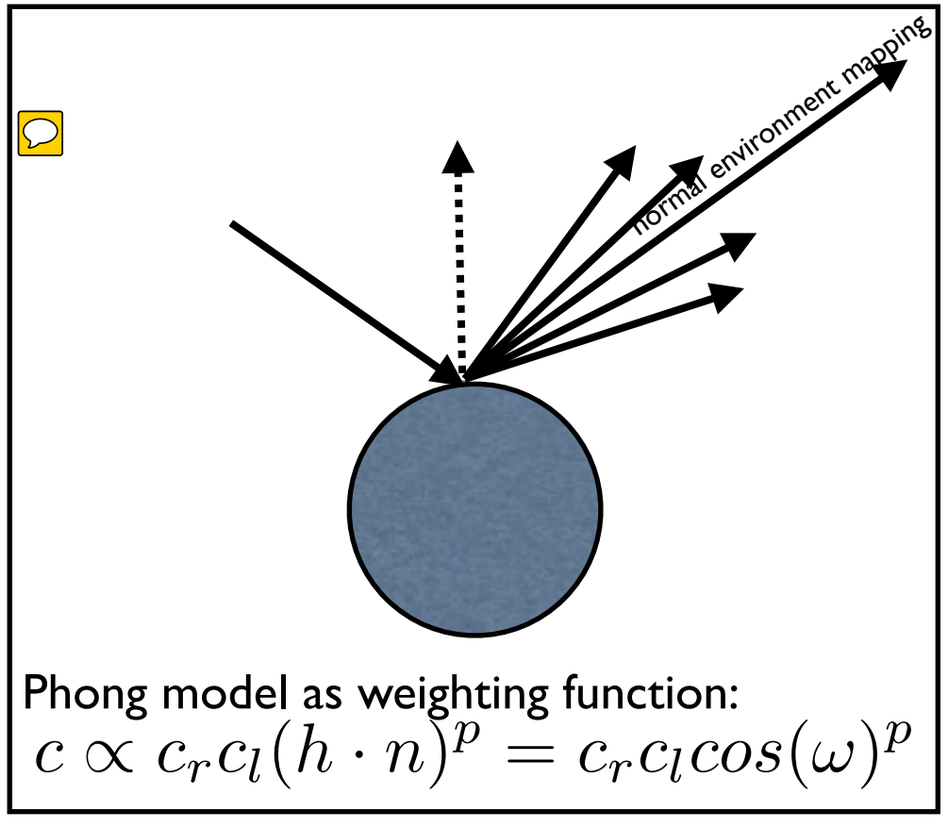
- Normal environment mapping allows to simulate perfectly shiny surfaces

- The textures of the environment map (EM) can be pre-filtered (eg. blurred to lead to rougher surface appearance - sometimes called reflection mapping)

environment map

- One way to do this, is to weight the texels of the EM based on the reflection direction, and look-up the EM for multiple reflection directions for each point

- Instead of applying a simple blur function (e.g., a Gauss kernel) to the EM, the reflection directions and their weights for each point can be computed with the Phong equation



Irradiance (Environment) Maps

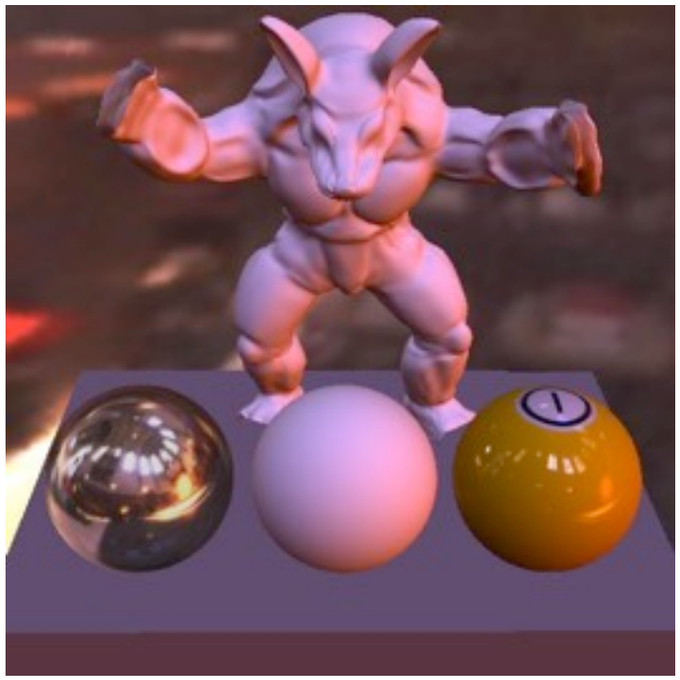
- Summing up the weighted contributions of a particular direction gives the diffuse lighting for that direction

- Doing this for all possible directions and storing the values in an environment map allows this EM to be used to simulate diffuse reflections from more complex environment irradiance

- This is called irradiance mapping (or irradiance environment mapping)

- It has the advantage of eliminating complex per-vertex lighting computations

- But, as for normal environment mapping, we have to assume that the environment lights are distant (ie. only perfectly correct for one point - the center point of the EM)



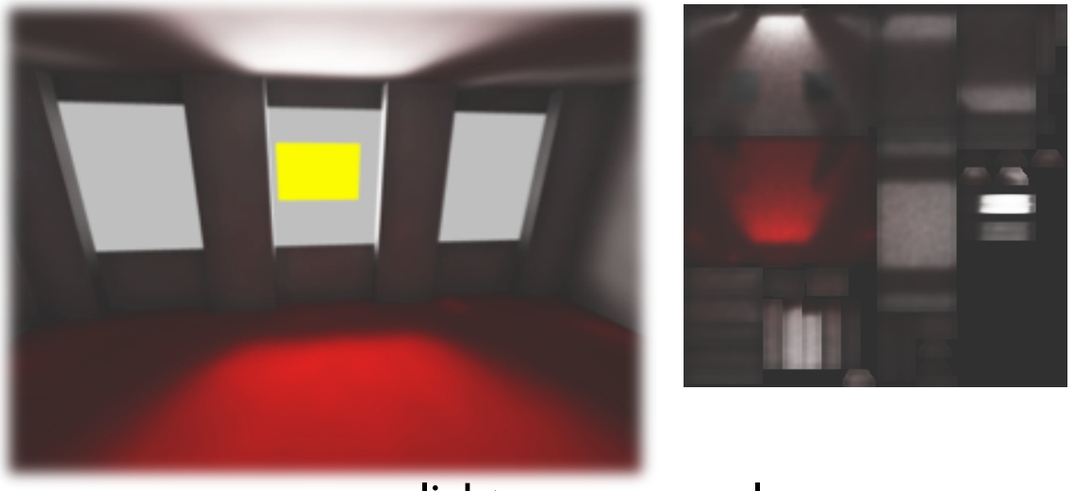
normal environment map

irradiance environment map

all as cube maps

Light Maps

- As radiosity is too slow for real-time global illumination, lighting can be precomputed and stored in textures
- These textures are called light maps
- They are used for static scenes and objects
- Assume static relationship of light and scene object
  - the light transport will be constant for static scenes
  - pre-computation of light transport result and store at per-vertex or per-texel (exitance value, irradiance value or irradiance direction)
    - view-independent effects (such as Lambertian diffuse reflection): store exitance value
    - view-dependent effects (such as normal map, mirror reflection): store irradiance value and direction, and calculate at run time (e.g., irradiance maps)
- Dynamic object are treated in additional

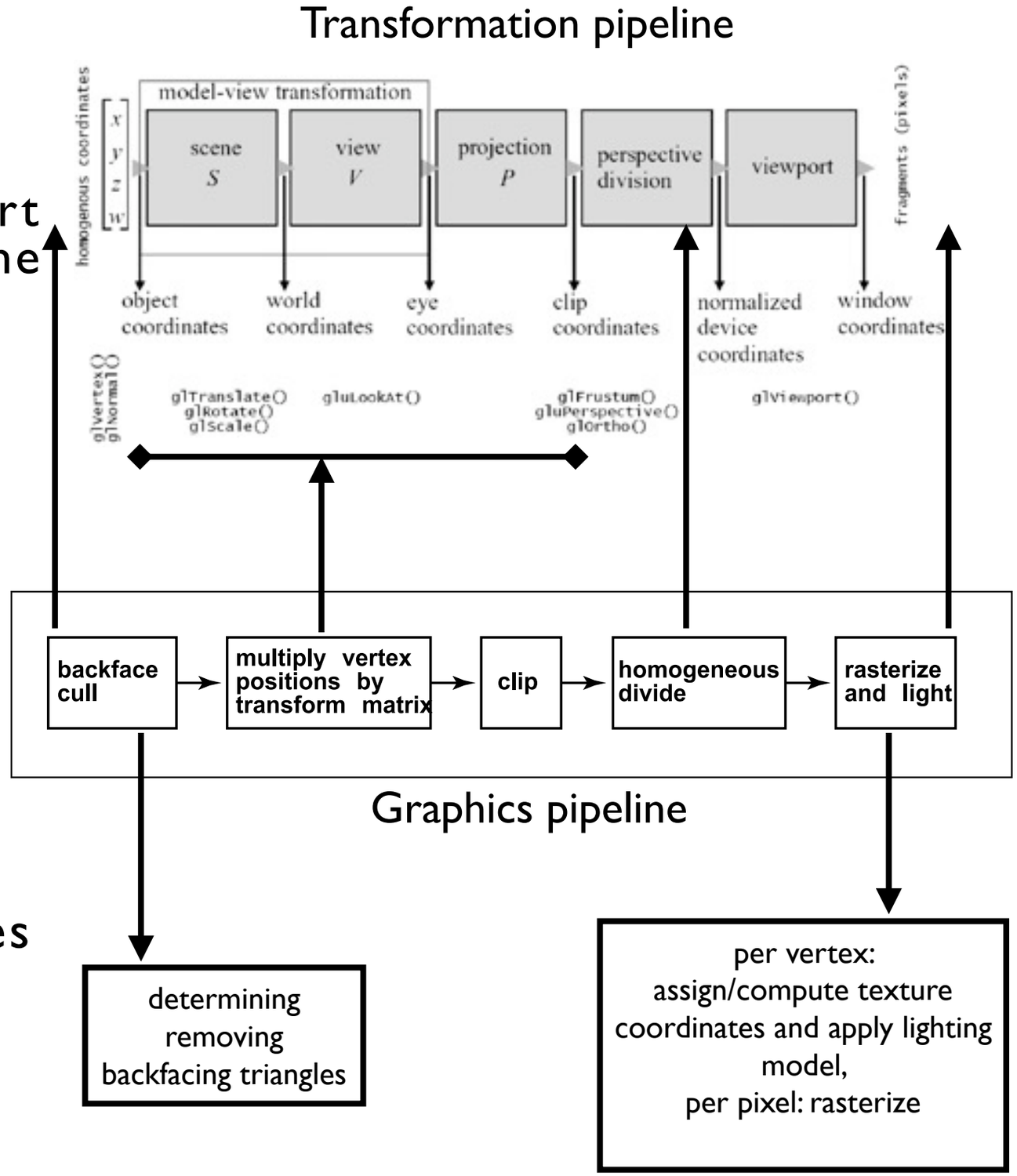


light map example

Graphics Pipelines

Putting Together The Bits and Pieces

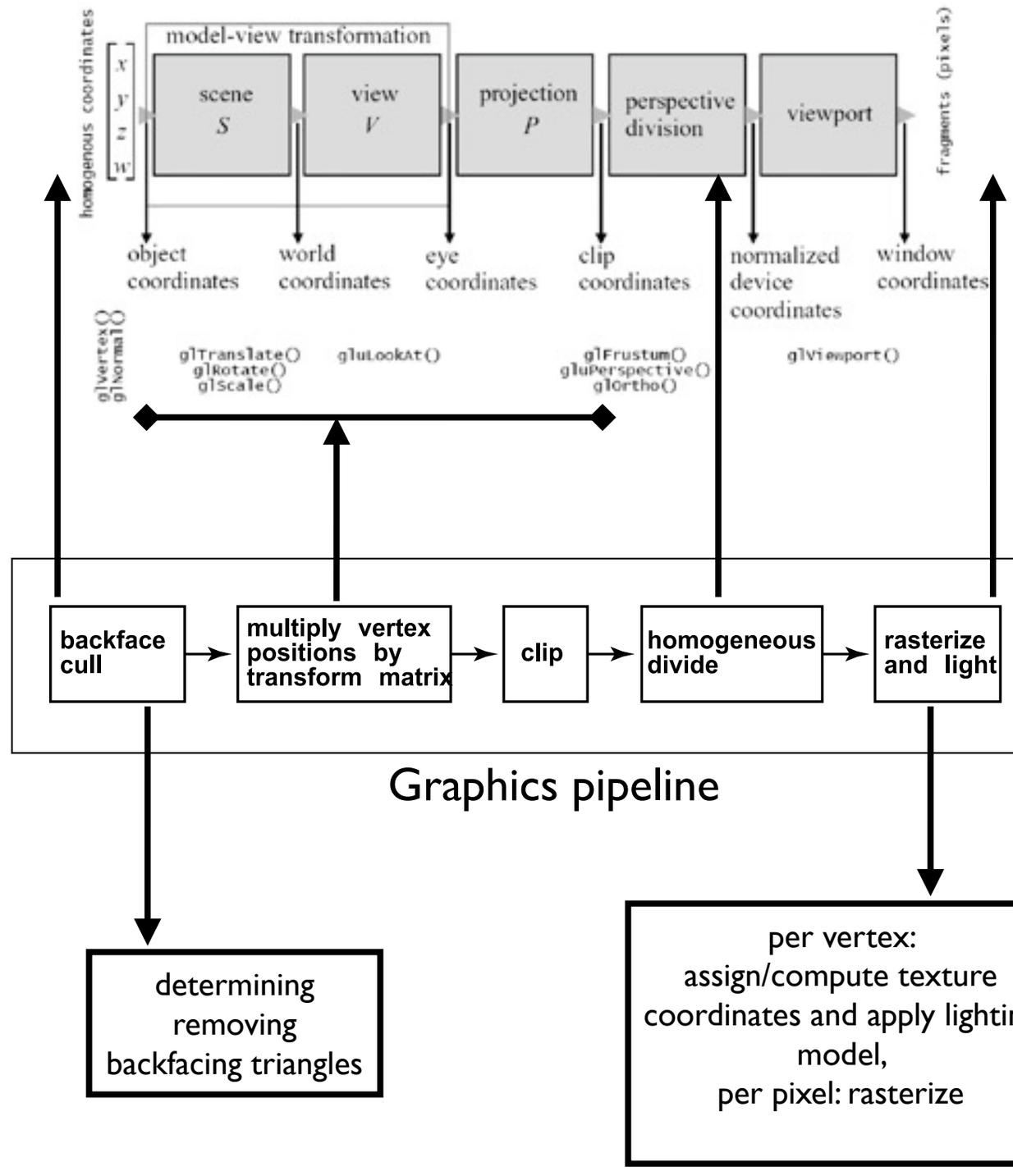
- For real-time graphics, we talked about:
  - Transformations (scene, view, projection, viewport) - this part is called transformation pipeline
  - Depth-handling, backface elimination/culling and rasterization
  - Lighting and shading
  - Texture mapping
- Putting these components together in the correct order leads us to what is called graphics pipeline
- The goal is to send as few triangles as possible through the graphics pipeline to achieve an optimal performance
- How?



Clipping vs. Culling

- When triangles are backfacing or are completely outside the viewing volume, then they are removed entirely from the pipeline (i.e., they are culled)
- If frontfacing triangles intersect the viewing volume, they are clipped
- In practice, culling can be very expensive, if carried out for each triangle individually
- Culling bounding volumes (spheres or boxes) are more efficient

Transformation pipeline



Notes on Backface Culling

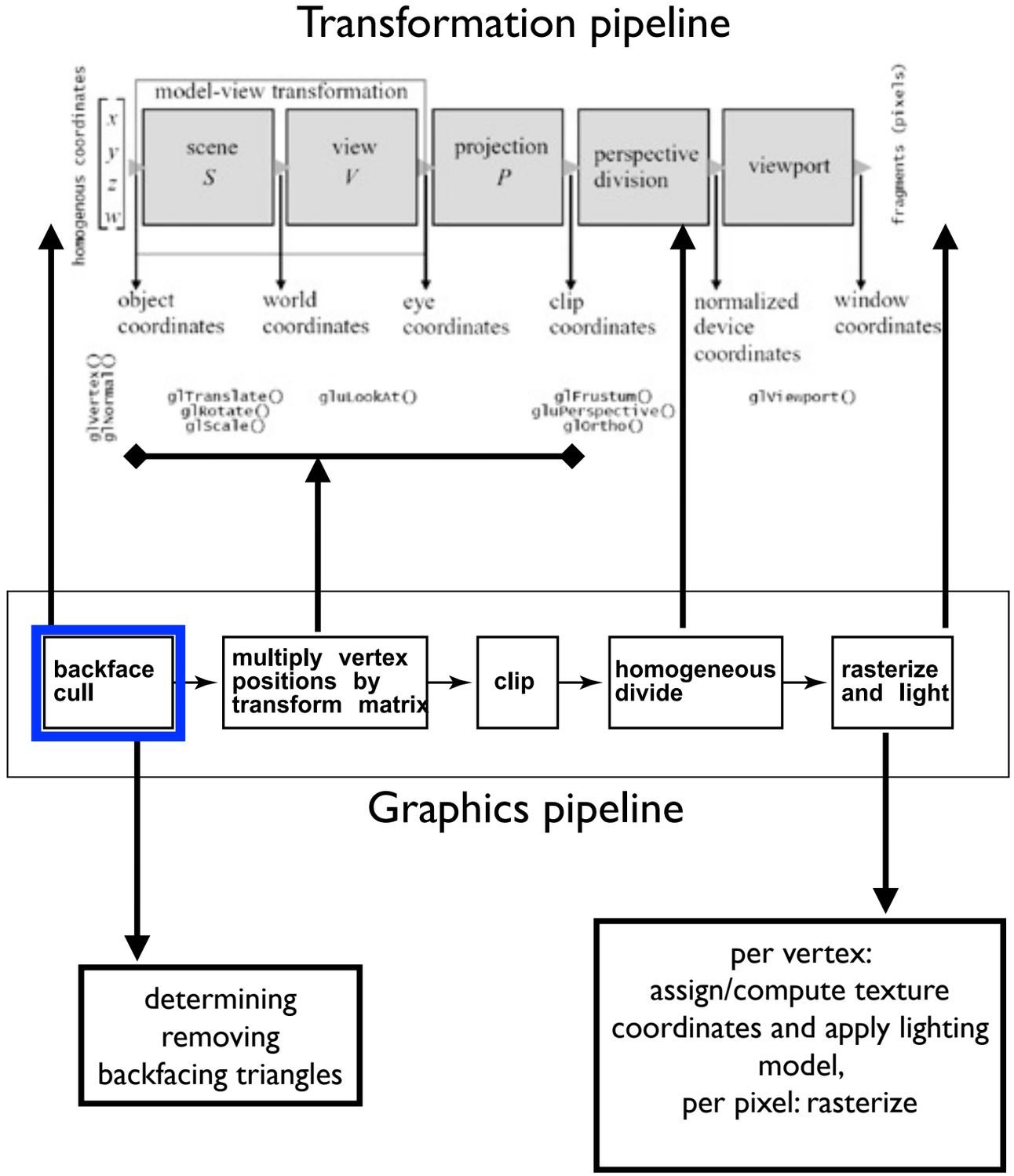
- This works well for closed surfaces - but not for open ones!

- In the latter case, triangles are made frontfacing and backfacing (by aligning two triangles with reverse normals - watch Z-buffer fighting!) or backface culling is turned off (this is bad for the performance of other closed surfaces in the scene)

- Good alternative (if supported):

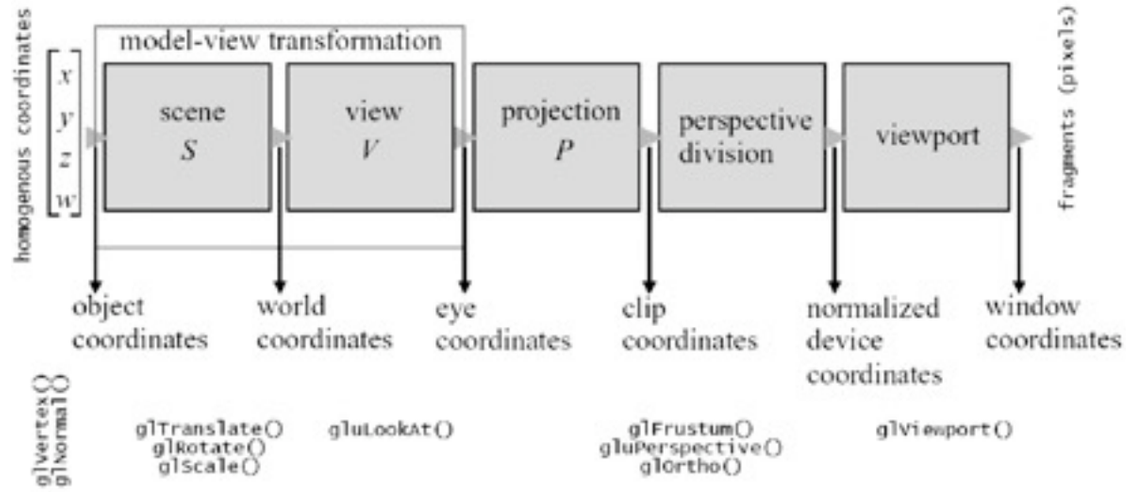
  1. Turn on backface culling and render closed surfaces

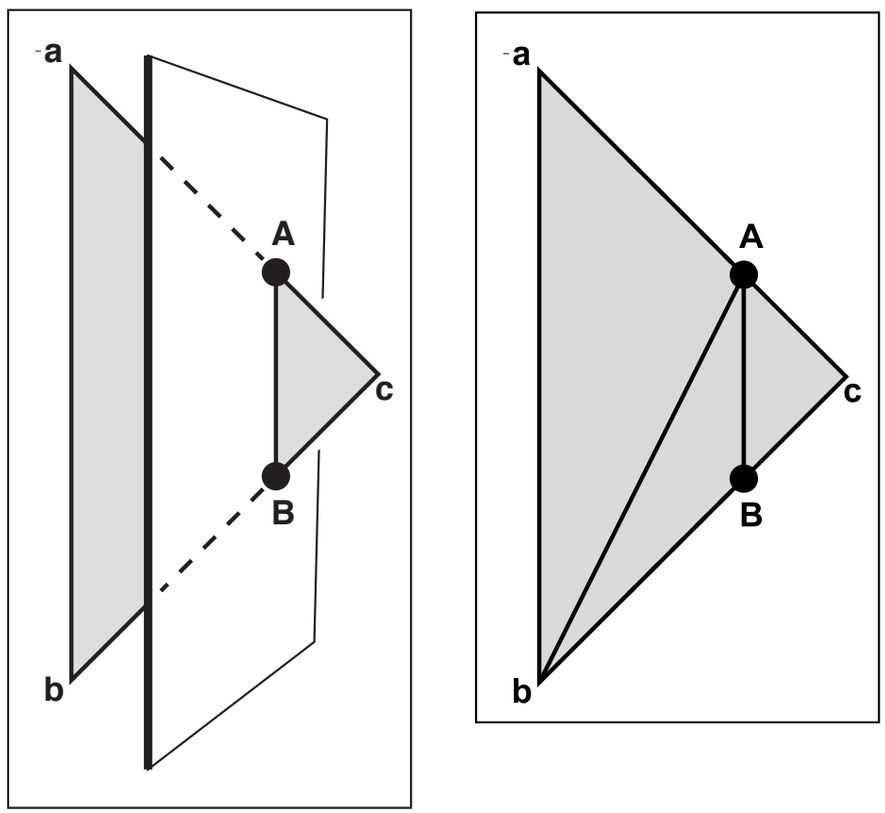
  2. Then turn off backface culling and render the rest



Clipping

- Clipping removes parts of triangles that intersect the viewing frustum and are (partially) outside
- At which state of the pipeline would you carry out clipping?

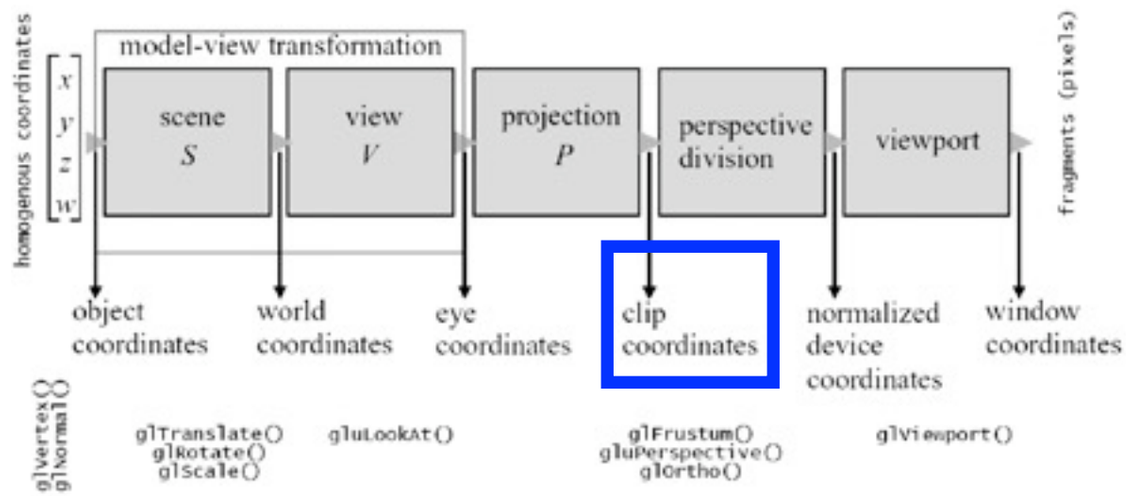


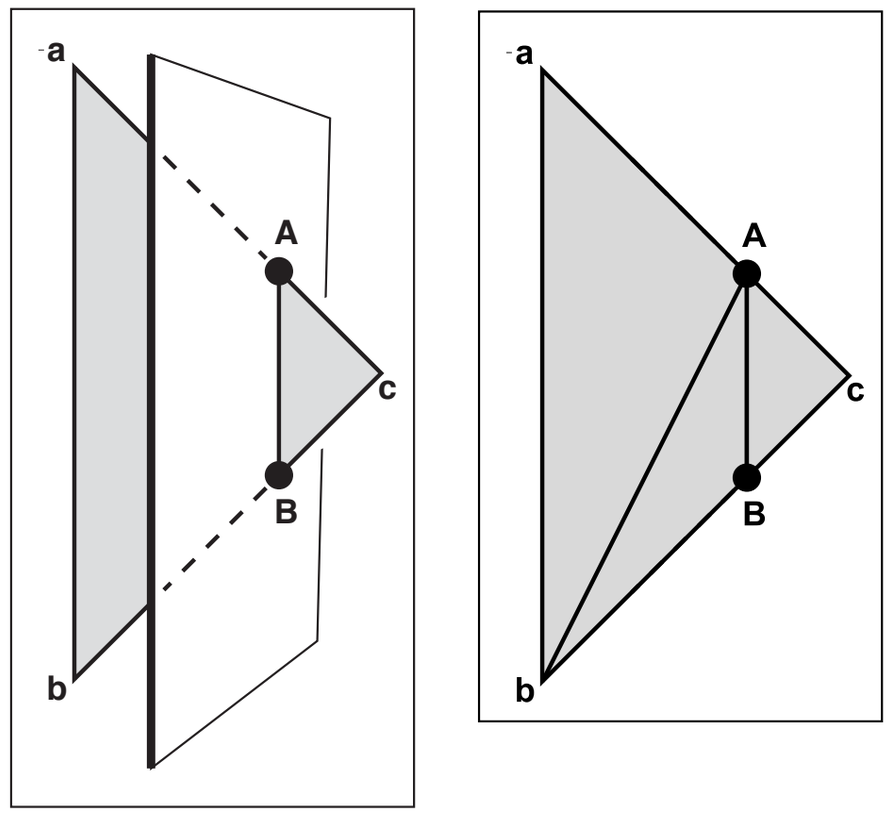


How triangles are handled that intersect a clipping plane was discussed in Depthhandling and Rasterization (BSP Trees)

Clipping

- Clipping removes parts of triangles that intersect the viewing frustum and are (partially) outside
- At which state of the pipeline would you carry out clipping?
- It can be applied after the triangles went through the transformation pipeline and are in camera coordinate system (e.g., before perspective division)
- That is why these coordinates are also called clip coordinates

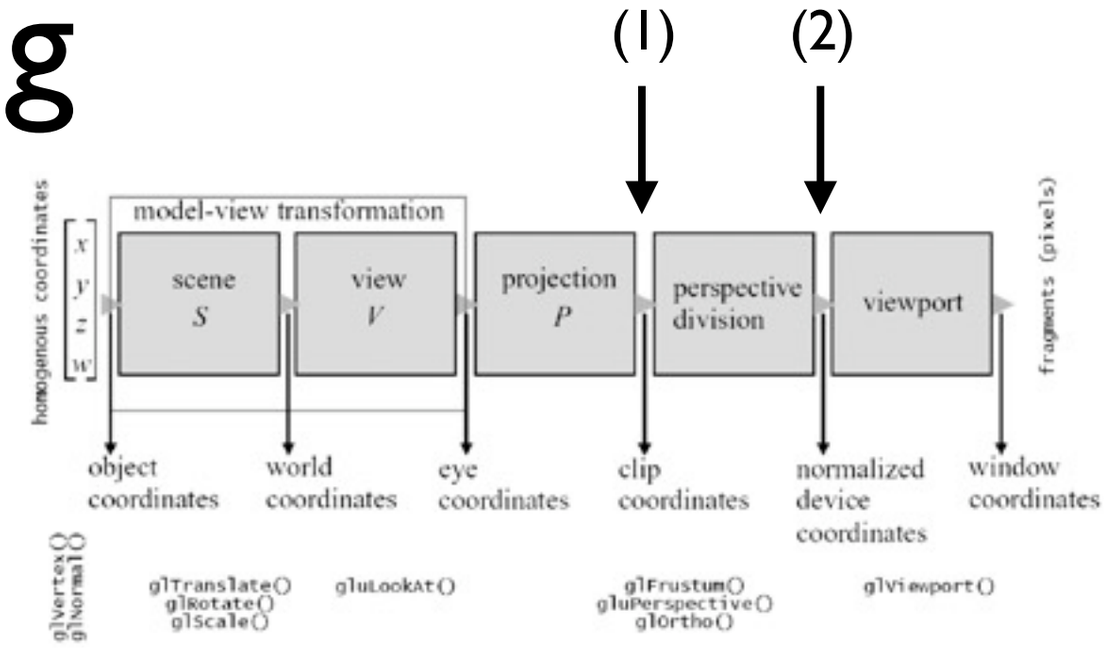




How triangles are handled that intersect a clipping plane was discussed in Depthhandling and Rasterization (BSP Trees)

Clipping

- But why is clipping carried out before perspective division (1), and not afterwards (2)?



Recap: Perspective Projection

- The value (f) is the distance to the far clipping plane of our viewing frustum, and (n) the distance to its near clipping plane
- Here, the homogeneous coordinate (h) plays an important role
- If we allow the homogeneous coordinate to take up any value (not just 0 or 1), then it can be used to encode how much the other three coordinates must be scaled after a perspective transform
- Dividing these three coordinates by the homogeneous coordinate is called homogenization or perspective division
- As for the orthographic case, the resulting value in the z component is not used yet - it will be used for hidden surface removal since it preserves depth order

Perspective transform matrix M_p

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ \boxed{z} \end{bmatrix}$$

homogeneous coordinate

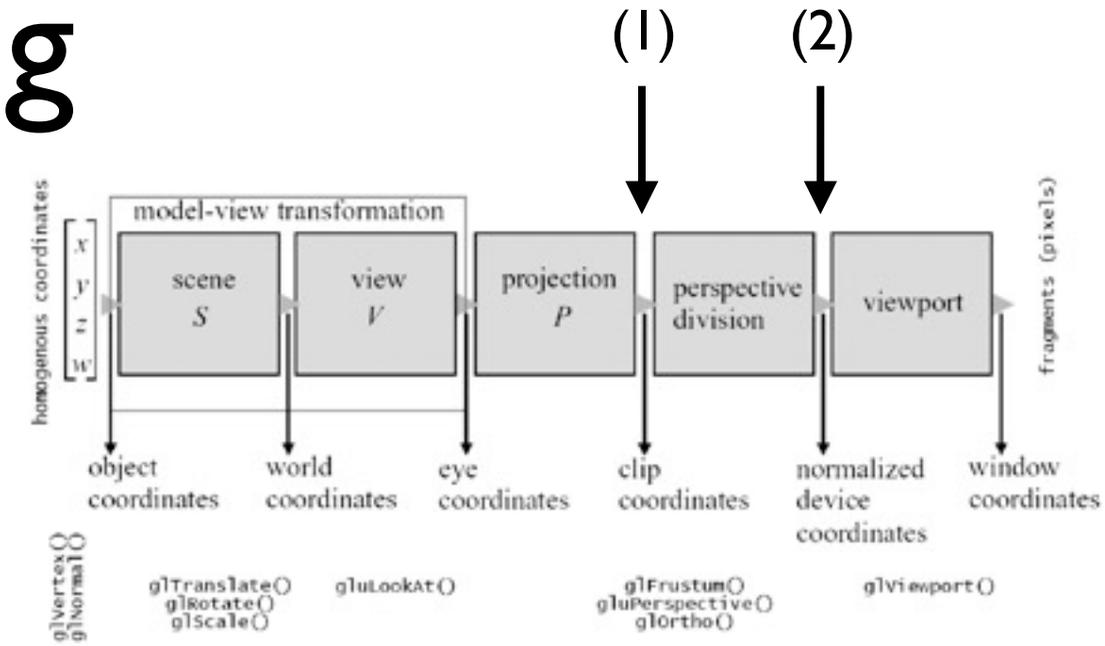
$$\begin{bmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{bmatrix} \boxed{/z} = \begin{bmatrix} \frac{n}{z}x \\ \frac{n}{z}y \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix}$$

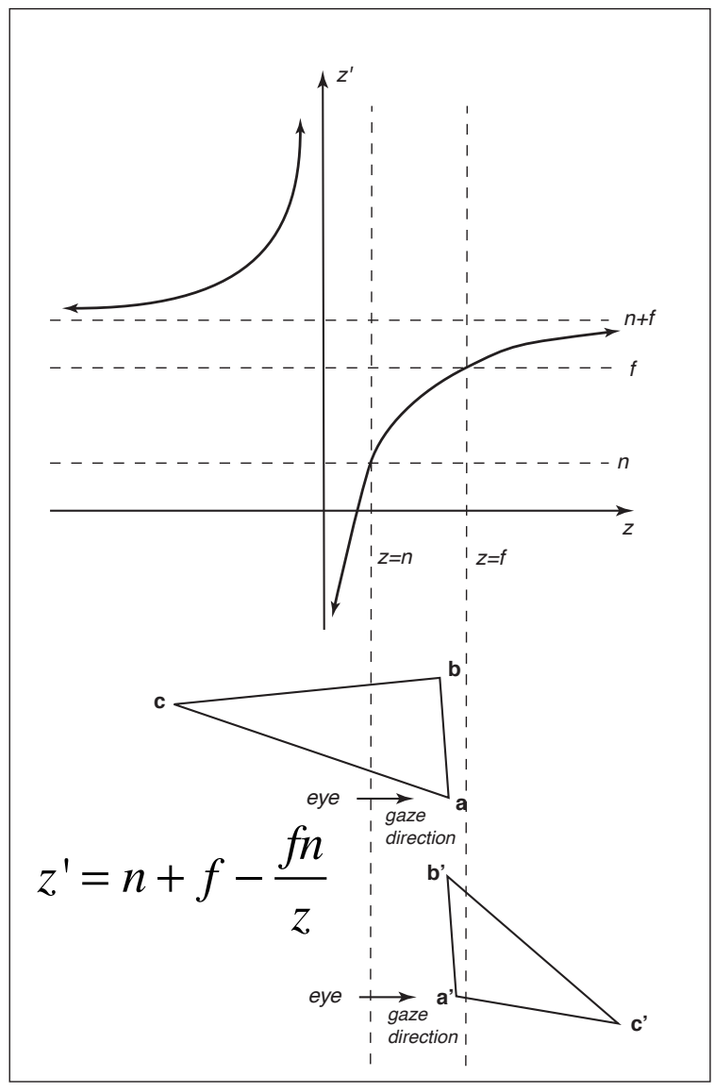
homogenization or perspective division

$$M_p^{-1} = \begin{bmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{bmatrix}$$

Clipping

- But why is clipping carried out before perspective division (1), and not afterwards (2)?
- Both is, in general, possible - as long as the correct plane equations are used (and the homogeneous coordinate w is considered appropriately)
- The problems for clipping with option (2) are the following:
  - The depth order is preserved after perspective division (good for Z-buffering) for depths greater than zero (z=0 is at the camera, negative z is behind the camera)
  - It is discontinuous at zero depth
  - When z moves from positive to negative values, z‘ switches from negative to positive



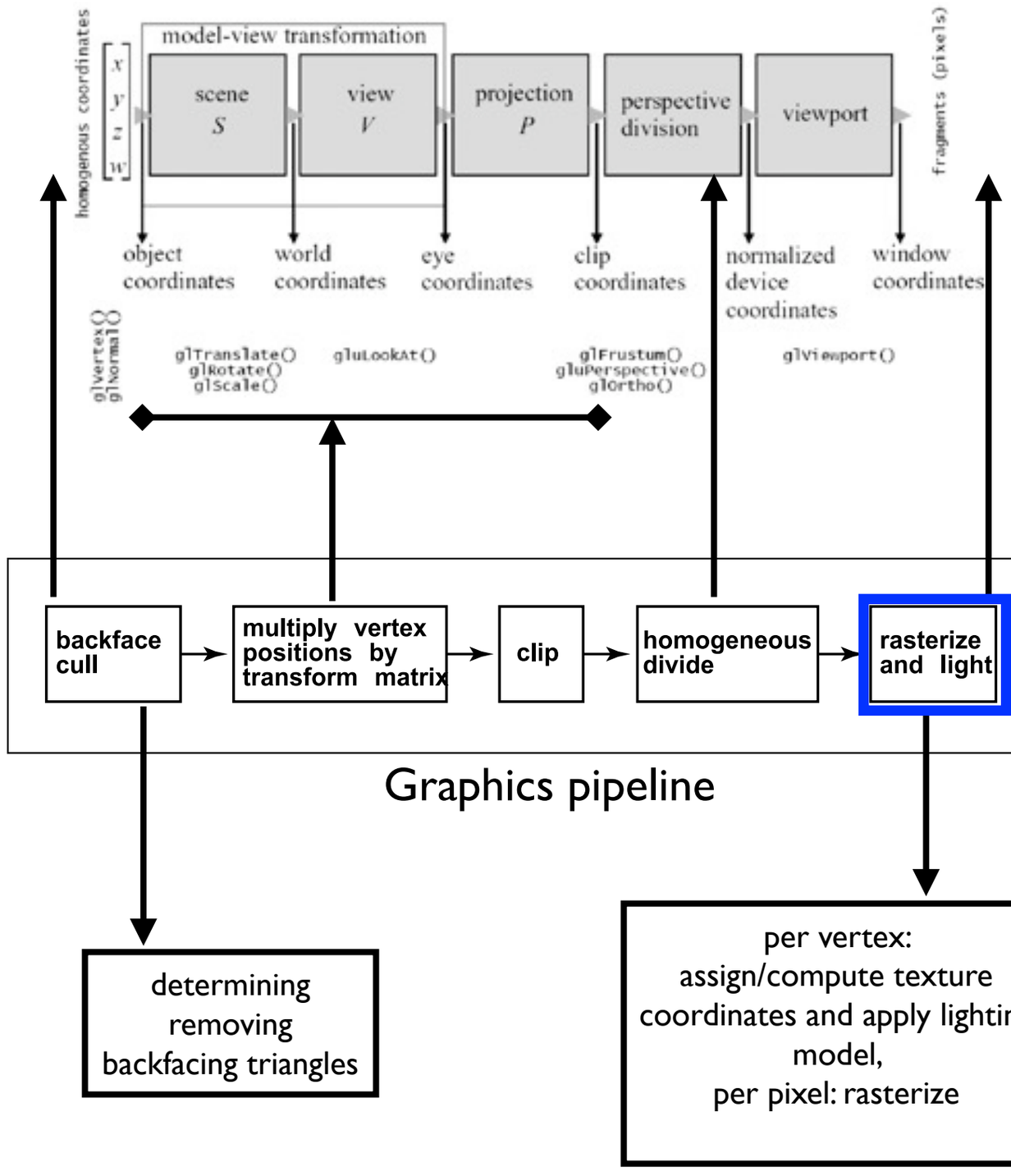


$$z' = n + f - \frac{fn}{z}$$

Notes on Lighting

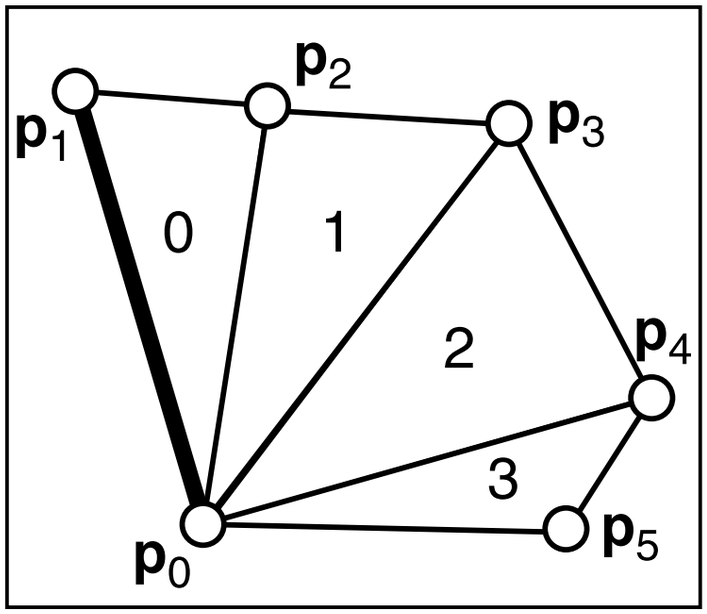
- In fact, lighting computations could also be carried out earlier (e.g., after scene transformation, when being in world coordinates)
- This has been done in traditional pipelines that support simple per-vertex shading models, like Gouraud shading
- But for more complex shading models, like Phong shading with normal interpolation, the shading has to be computed on a per-pixel basis (i.e., during rasterization) - this is the trend in modern pipelines

Transformation pipeline



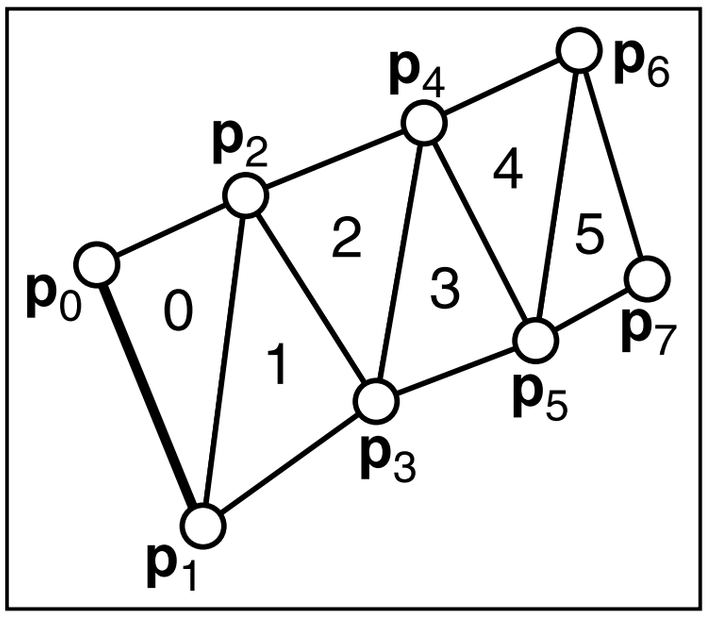
Triangle Strips and Fans

- The rendering performance can not only be increased by reducing the number of triangles, but also by reducing the number of vertices
- Surfaces usually consist of a number of connected triangles
- If triangles are aligned, they share vertices



Triangle fan

- It does not make sense to process vertices with the same coordinates multiple times
- Therefore, graphics pipelines support efficient handling of specific mesh structures, such as fans and strips
- In practice, speed-ups of factor 3 can be reached when using triangle strips compared to single triangles



Triangle strip

Preserved States

- Having to set the attributes (e.g., color or material properties, etc.) for each triangle and each vertex individually will cost a significant amount of time

- That is why many pipelines support sharing these attributes and states among many triangles (remember: OpenGL, for instance, is a state machine) - this can result in considerable speed-ups

- Another state that can be saved is the geometry itself
  - This is beneficial if geometry is static and does not change from frame to frame
  - It is stored in so called display lists
  - Display lists are efficient because they are stored on the graphics card (i.e., they don‘t have to be continuously transferred over the bus), and they can be automatically optimized (e.g., triangles that share the same vertices are grouped so that they can live in the same set of calls with shared atributes)

Without state-machine:

set triangle atributes T1
draw triangle T1
set triangle atribute T2
draw triangle T2
...

With state-machine:

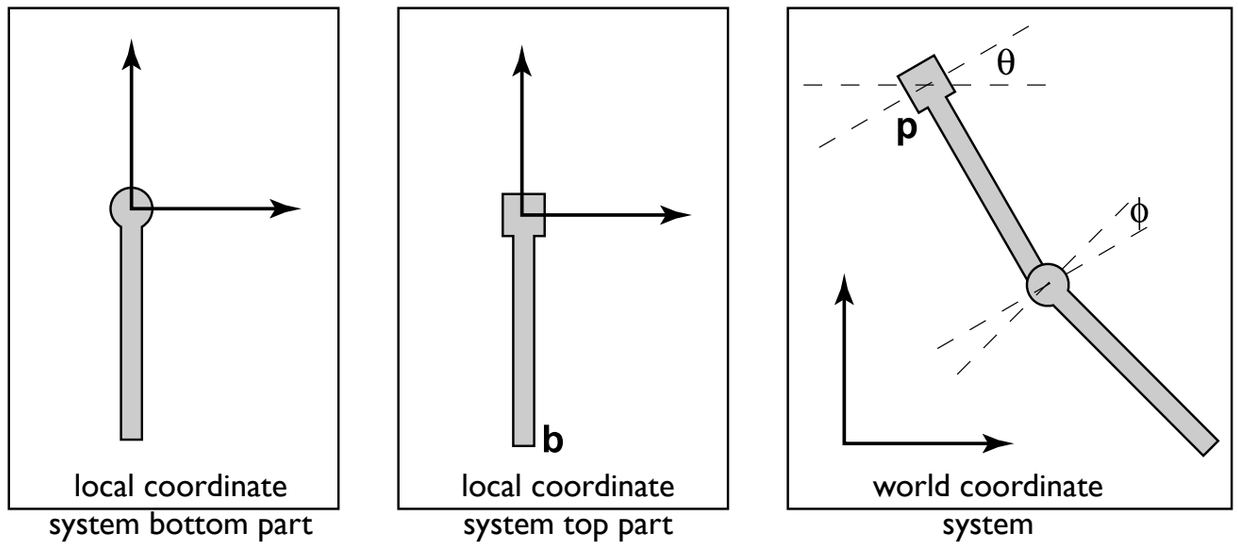
set state triangle atributes
draw triangle T1
draw triangle T2
...

Encapsulated Transformations

- Let‘s assume you wanted to implement more complex, encapsulated transformations of connected components, such as the pendulum on the right side

- We can assume that each component is transformed in its own local coordinate system, and that the local coordinate systems are transformed with respect to the hierarchical connectivities of the components

Pendulum example



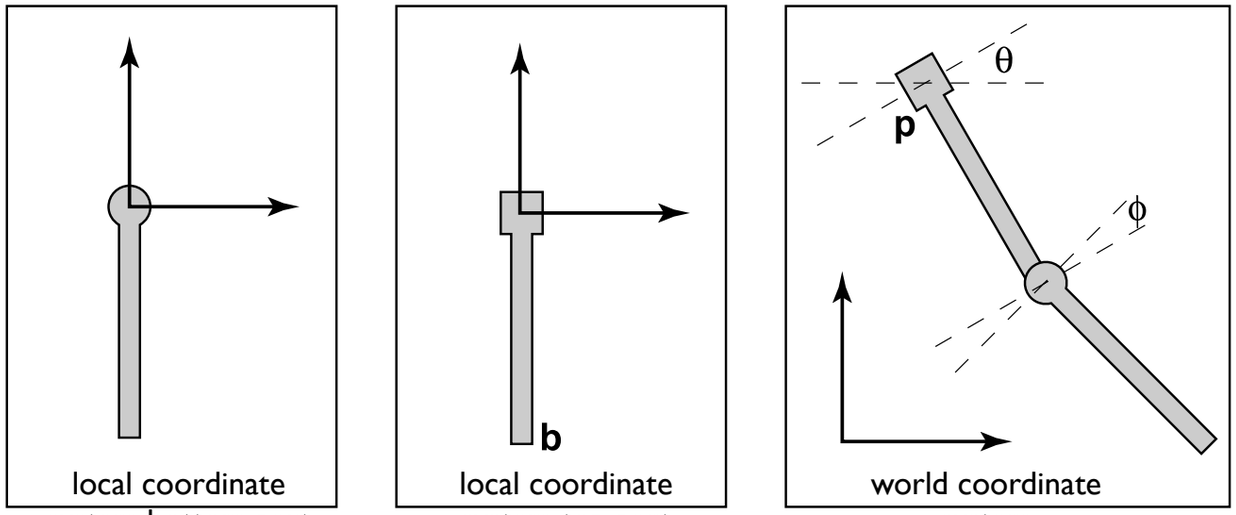
Encapsulated Transformations

- Let‘s assume you wanted to implement more complex, encapsulated transformations of connected components, such as the pendulum on the right side

- We can assume that each component is transformed in its own local coordinate system, and that the local coordinate systems are transformed with respect to the hierarchical connectivities of the components

- For example: the upper part of the pendulum is transformed in its local coordinate system (this leads to M_3), and the lower part is transformed in its own local coordinate system (this leads to M_c) - since the lower part depends on the upper part, the lower coordinate system has to be moved with the upper one ($M_d=M_3M_c$)

Pendulum example



Top part:

$\mathbf{M_1}$=rotate(θ)
$\mathbf{M_2}$=translate(p)
$\mathbf{M_3}=\mathbf{M_2M_1}$
apply $\mathbf{M_3}$ to all points of upper part

Bottom part:

$\mathbf{M_a}$=rotate(ϕ)
$\mathbf{M_b}$=translate(b)
$\mathbf{M_c}=\mathbf{M_bM_a}$
$\mathbf{M_d}=\mathbf{M_3M_c}$
apply $\mathbf{M_d}$ to all points of lower part

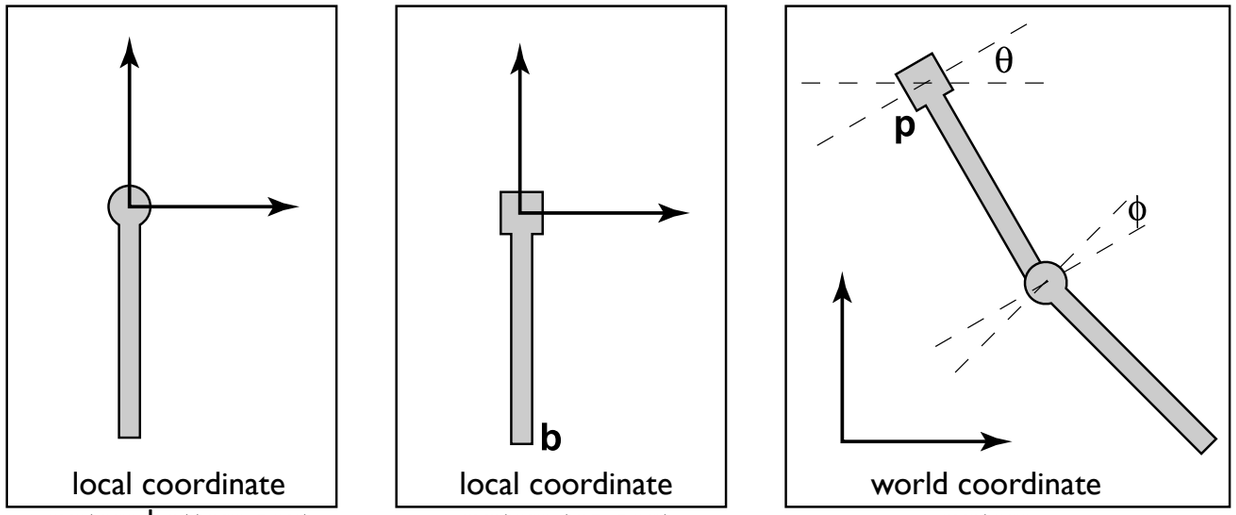
Encapsulated Transformations

- Let‘s assume you wanted to implement more complex, encapsulated transformations of connected components, such as the pendulum on the right side

- We can assume that each component is transformed in its own local coordinate system, and that the local coordinate systems are transformed with respect to the hierarchical connectivities of the components

- For example: the upper part of the pendulum is transformed in its local coordinate system (this leads to M_3), and the lower part is transformed in its own local coordinate system (this leads to M_c) - since the lower part depends on the upper part, the lower coordinate system has to be moved with the upper one ($M_d=M_3M_c$)

Pendulum example



Top part:

$\mathbf{M_1}$=rotate(θ)
$\mathbf{M_2}$=translate(p)
$\mathbf{M_3}=\mathbf{M_2M_3}$

apply $\mathbf{M_3}$ to all points of upper part

transformations in local coordinate systems

Bottom part:

$\mathbf{M_a}$=rotate(ϕ)
$\mathbf{M_b}$=translate(b)
$\mathbf{M_c}=\mathbf{M_bM_a}$
$\mathbf{M_d}=\mathbf{M_3M_c}$

hierarchical update

apply $\mathbf{M_d}$ to all points of lower part

Encapsulated Transformations

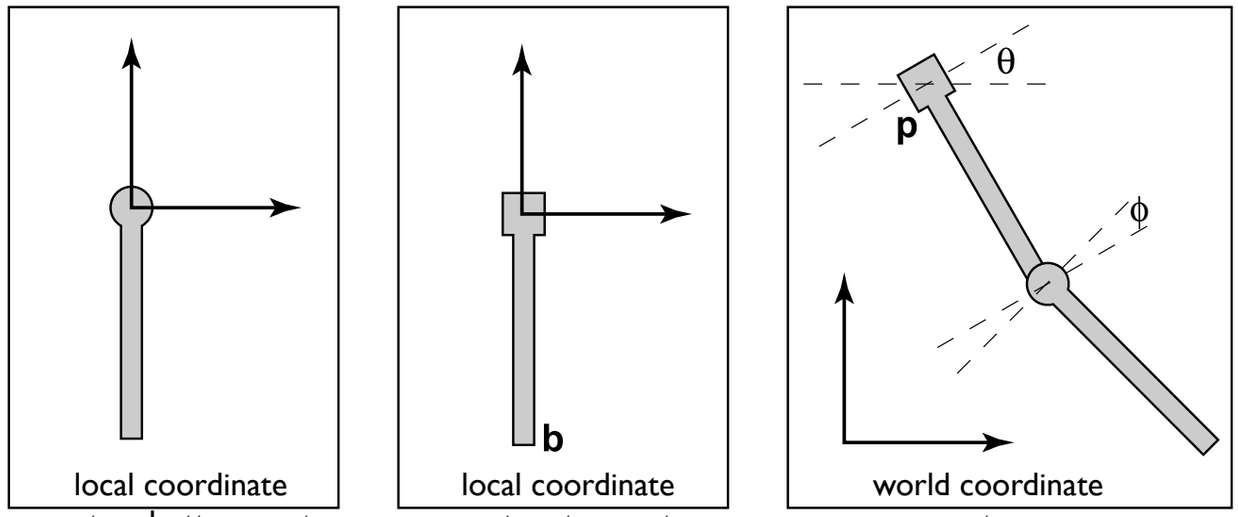
- This leads to a hierarchical transformation structure that manages all these transformations and local coordinates easily

- The correct transformation that is required for a particular object in this data structure is simply the product of all transformations on the way from the root node to the object node (previous * next !)

- The individual transformations within each node, are local transformations in the objects' local coordinate systems

- Accumulating all transformations from the root note to the object node leads to a composite transformation in the world coordinate system

Pendulum example



Top part:

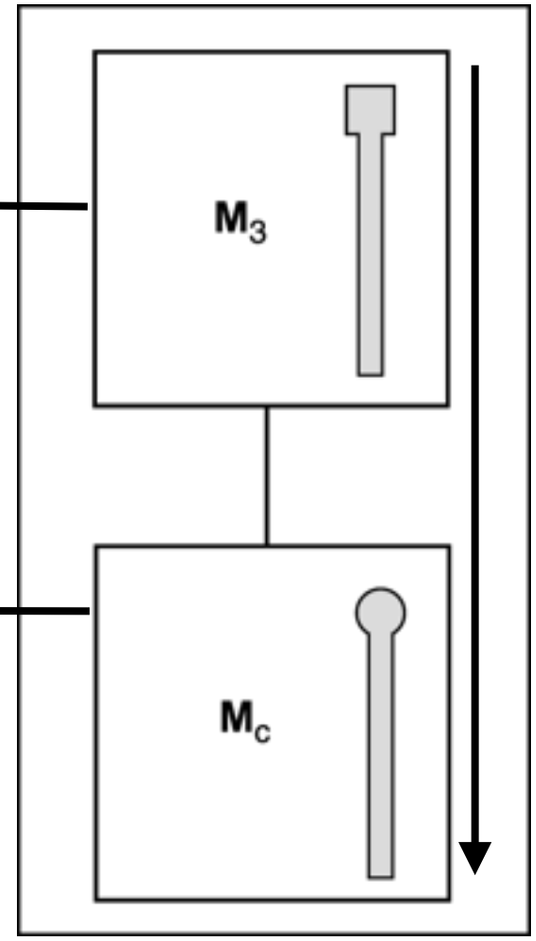
$\mathbf{M_1}$=rotate(θ)
$\mathbf{M_2}$=translate(p)
$\mathbf{M_3}=\mathbf{M_2}\mathbf{M_3}$

apply $\mathbf{M_3}$ to all points of upper part

Bottom part:

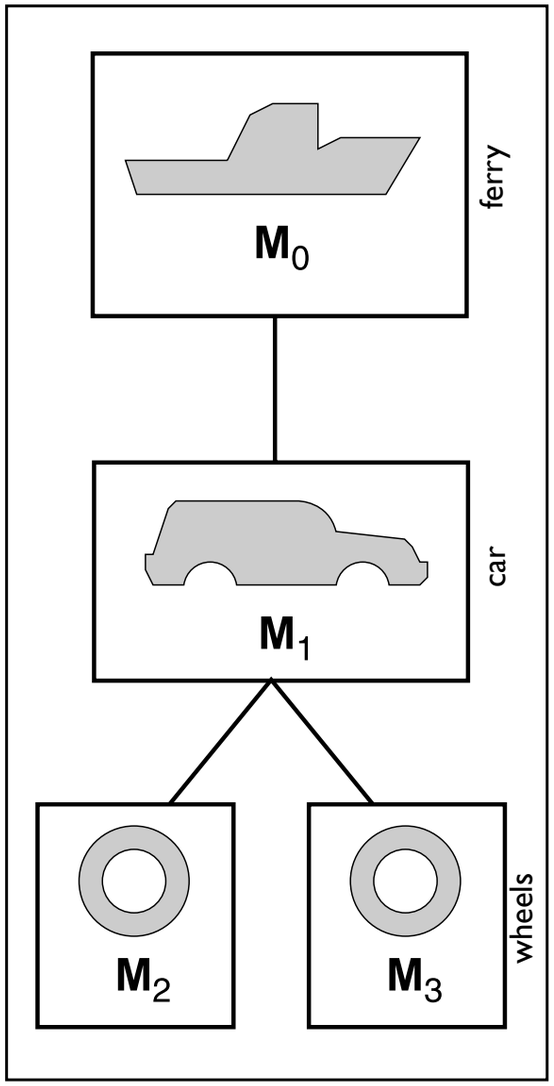
$\mathbf{M_a}$=rotate(ϕ)
$\mathbf{M_b}$=translate(b)
$\mathbf{M_c}=\mathbf{M_b}\mathbf{M_a}$
$\mathbf{M_d}=\mathbf{M_3}\mathbf{M_c}$

apply $\mathbf{M_d}$ to all points of lower part



Scene Graphs Basics

- This hierarchical data structure can be arbitrarily complex - with multiple child nodes
- It is a tree-like structure called scene graph
- Yet, the principle is always the same: individual transformations within nodes are transformations in local object coordinate systems - accumulating them top down leads to the composite transformation in the world coordinate system
- Scene graphs are efficiently implemented using matrix stacks
  - push() and pop() add and delete matrices from the right-hand side of the matrix product
  - they can be used for traversing a scene graph



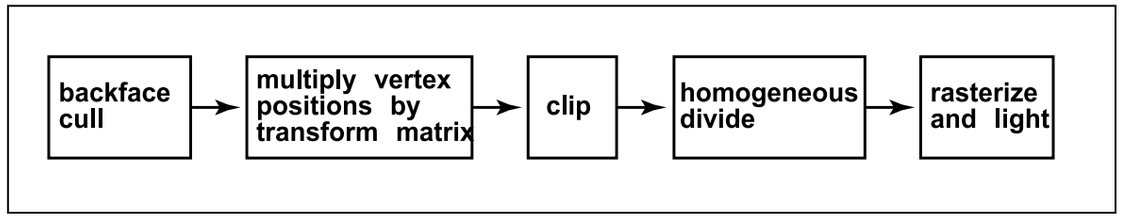
Example:
transformation of right wheel in world coordinates is $M_0M_1M_3$ and of left wheel $M_0M_1M_2$

Example:
push(M_0)
push(M_1)
push(M_2)
leads to $M_0M_1M_2$

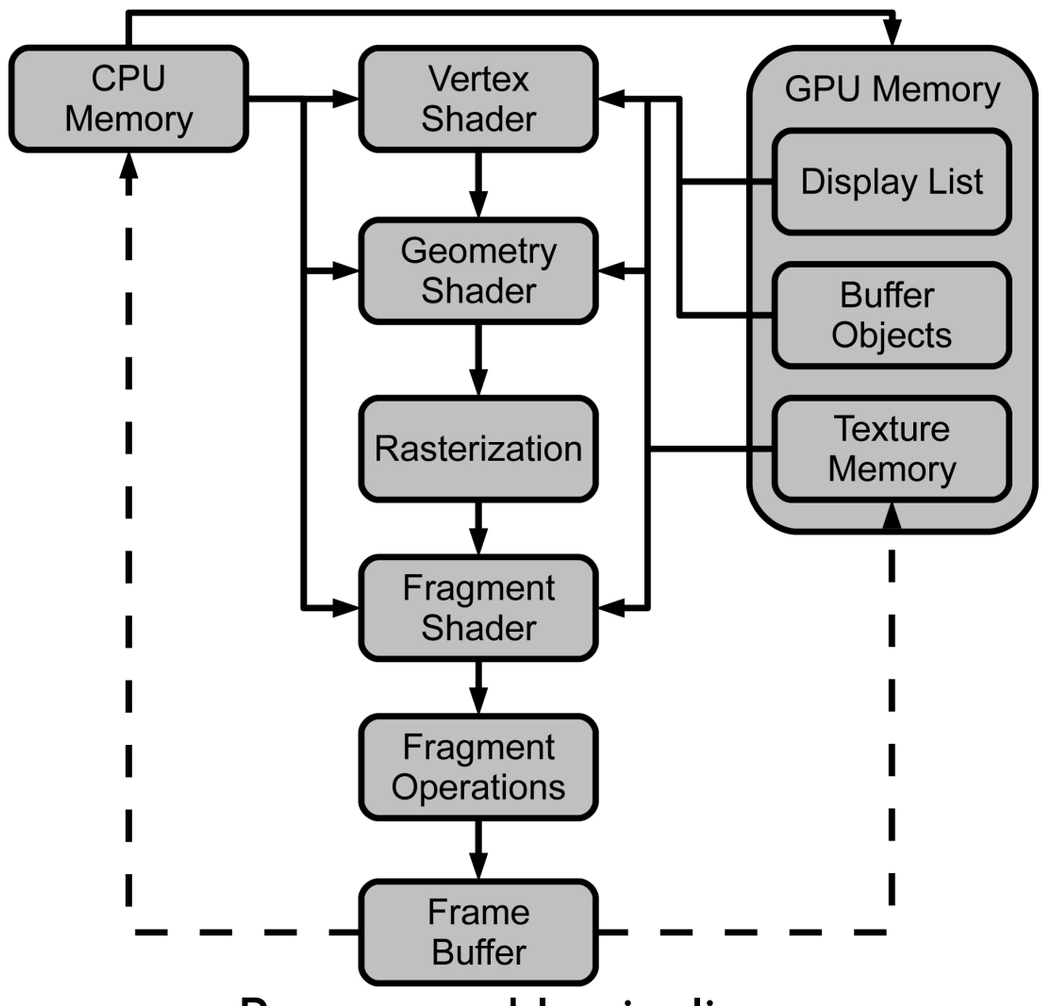
traverse(node)
  push(M_{local})
  draw(object using composite matrix from stack)
  traverse(left child)
  traverse(right child)
  pop()

Fixed Function vs. Programmable

- But for shared states we also have to pay a toll: flexibility!
- Traditional pipelines share many states (e.g., transformations, lighting parameters, etc)
- This is also the case for classical OpenGL - these pipeline are called fixed function pipelines
- Many states (such as transformations) are the same for all vertices in fixed function pipelines (i.e., they cannot be changed during rendering one primitive)
- Todays programmable pipelines allow individual vertex, fragment and geometry operations - with shaders
- Flexible texture access is possible in all shaders



Fixed function pipeline



Programmable pipeline

NEXT ICG LAB TALK:
26. APRIL 2016, 4:00PM

Institute of Computer Graphics

Prof. Daniel Weiskopf

University of Stuttgart

Eye Tracking and Visualization

Computer Science Building (S3)

Room S3 048

JKU

For more information about our talks visit
http://www.cg.jku.at/talks/invited

Course Schedule

Type	Date	Time	Room	Topic	Comment
C1	01.03.2016	13:45-15:15	HS 18	Introduction and Course Overview	Conference
C2	15.03.2016	13:45-15:15	HS 18	Transformations and Projections	Easter Break
C3	05.04.2016	13:45-15:15	HS 18	Raster Algorithms and Depth Handling	
C4	12.04.2016	13:45-15:15	HS 18	Local Shading and Illumination	
C5	19.04.2016	13:45-15:15	HS 18	Texture Mapping Basics	
C6	26.4.2016	13:45-15:15	HS 18	Advanced Texture Mapping & Graphics Pipelines	
C7	03.05.2016	13:45-15:15	HS 18	Intermediate Exam	
C8	09.05.2016	17:15-18:45	HS 18	Global Illumination I: Raytracing	
C9	10.05.2016	13:45-15:15	HS 18	Global Illumination II: Radiosity	Conference / Holiday
C10	31.05.2016	13:45-15:15	HS 18	Volume Rendering	
C11	07.06.2016	13:45-15:15	HS 18	Scientific Data Visualization	
C12	14.06.2016	13:45-15:15	HS 18	Curves and Surfaces	
C13	21.06.2016	13:45-15:15	HS 18	Basics of Animation	
C14	28.06.2016	13:45-15:15	HS 18	Final Exam	
C15	04.10.2016	13:45-15:15	TBA	Retry Exam	

Thank You!