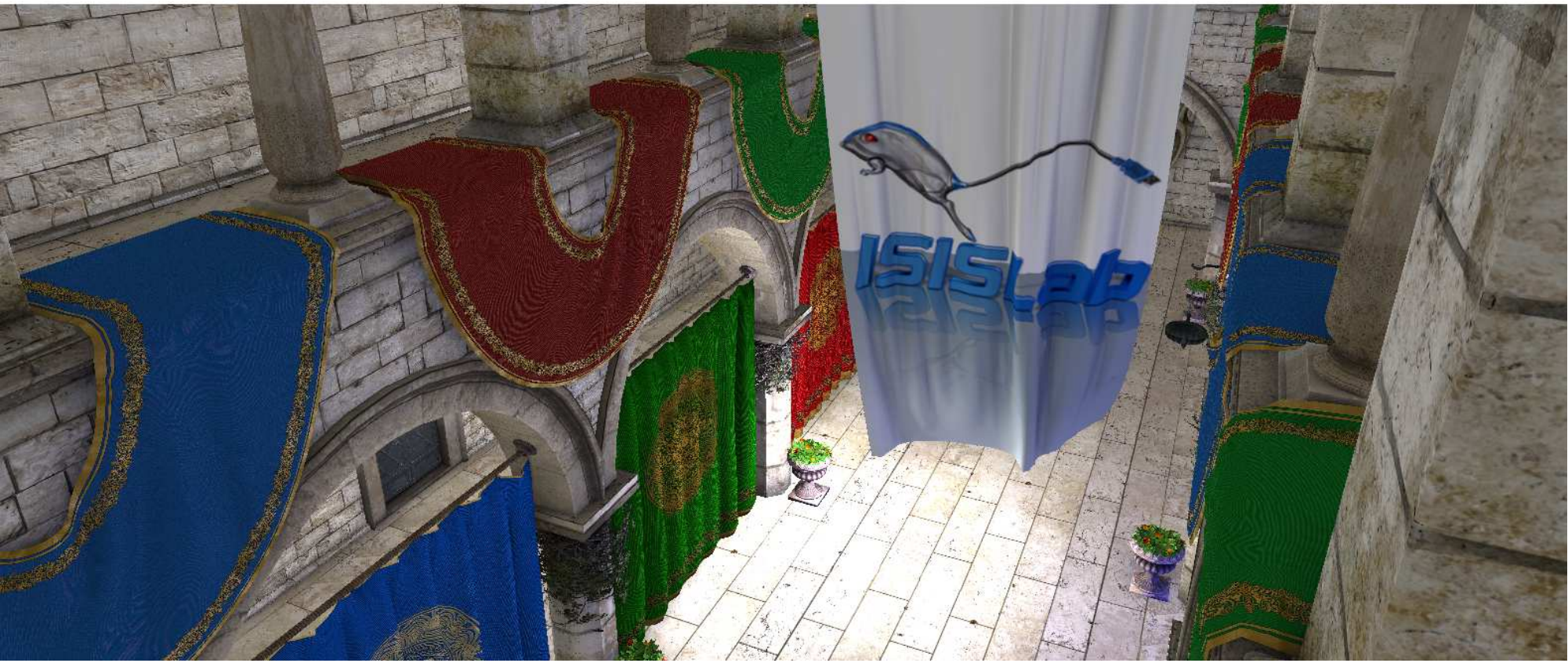
OpenGL fun in the Sponza Atrium



Dario Scarpa - dario.scarpa@duskzone.it

ISISlab
Università degli studi di Salerno - Facoltà di Scienze MM. FF. NN.

Outline

1. Introduction
2. A modern OpenGL sandbox
3. Forward VS Deferred Rendering
4. Disk Based SSAO strikes back
5. Screen Space Global Illumination
6. Conclusions

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering

4 Disk Based SSAO strikes back

5 Screen Space Global Illumination

6 Conclusions

Goal

- evaluating and experimenting the applicability of the approach used in **Disk Based Screen Space Ambient Occlusion** to approximate **Global Illumination** in **real-time rendering**

Goal

- evaluating and experimenting the applicability of the approach used in **Disk Based Screen Space Ambient Occlusion** to approximate **Global Illumination** in **real-time rendering**
- ...what?

Side goals

- getting basic computer graphics theory
  - 3D transformations, physics of light...
- learn low level graphics programming
  - modern OpenGL, GPU architecture

Side goals

- getting basic computer graphics theory
  - 3D transformations, physics of light...
- learn low level graphics programming
  - modern OpenGL, GPU architecture
- understanding more John Carmack tweets

- master degree
  - someday

Rendering

- the process of producing a 2D image from a description of a 3D scene

Rendering

- the process of producing a 2D image from a description of a 3D scene
- ubiquitous: movies, games, visualization
- including ideas from a broad range of disciplines
  - lots of algorithms (and data structures)
  - applied physics and maths
  - ...but also biology, psicology and study of perception

Offline vs real-time rendering

- hours vs milliseconds
  - drastically different techniques are needed
- **offline**: physically based simulation, photorealism
  - mostly used in movies
- **real-time**: what matters is a visually pleasing image
  - used in interactive applications (videogames, VR)

Offline vs real-time rendering

- huge technological evolution
  - GPU parallel **processing power** increases
  - GPU **programmablity** gives programmers a huge creative freedom

- more and more rendering techniques inspired by physically based offline rendering enter the real-time rendering domain

Rendering and Illumination

Deep down, it's all about **setting pixel colors**

- color is defined by the interaction between light and matter
- so we need to model and approximate the behavior of light in the real world
  - or define a non-photorealistic behavior ourselves

Rendering and Illumination

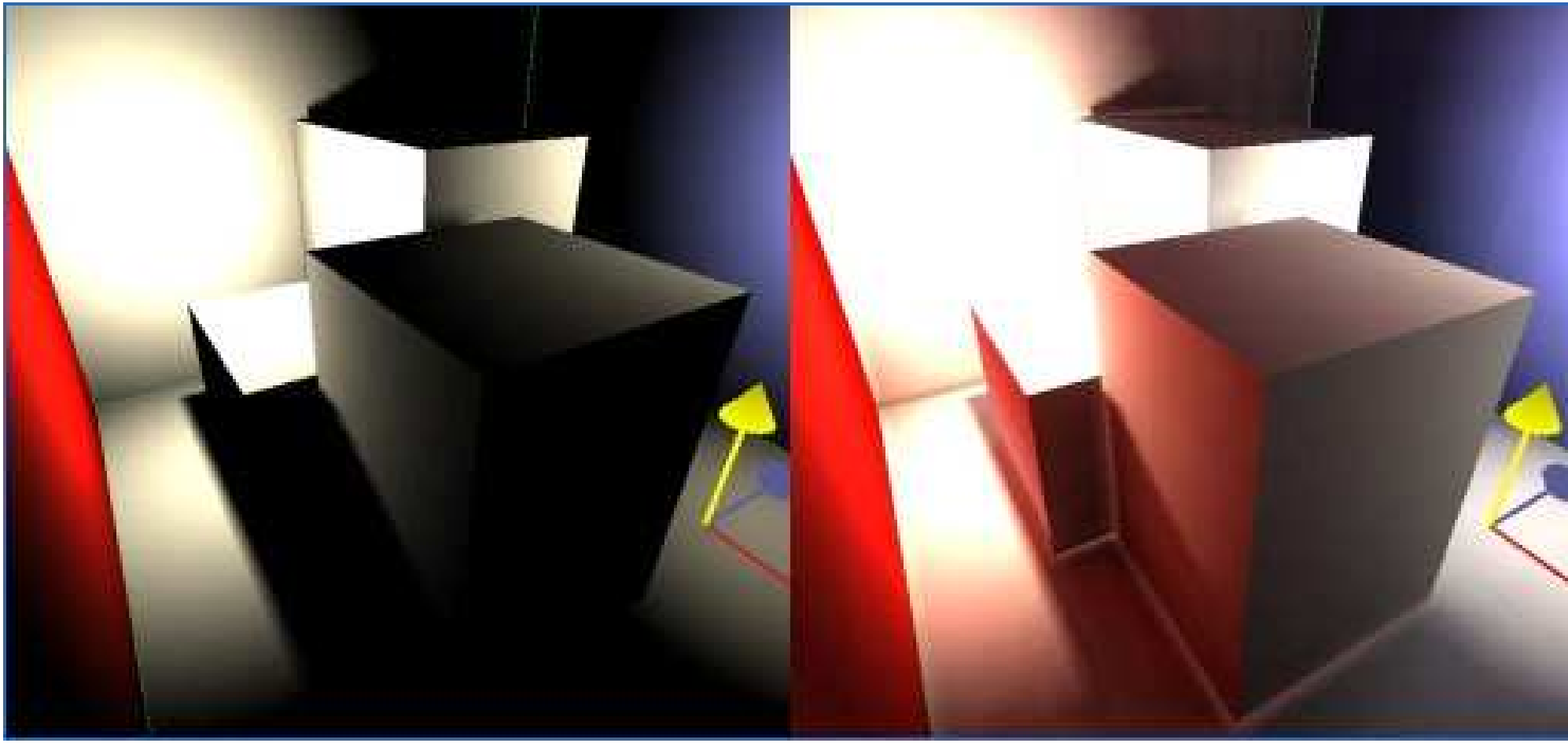
Deep down, it's all about **setting pixel colors**

- color is defined by the interaction between light and matter
- so we need to model and approximate the behavior of light in the real world
  - or define a non-photorealistic behavior ourselves

Illumination is **the** problem of computer graphics

- An **illumination model** is a mathematical function that models how a surface interacts with light

Local VS Global Illumination

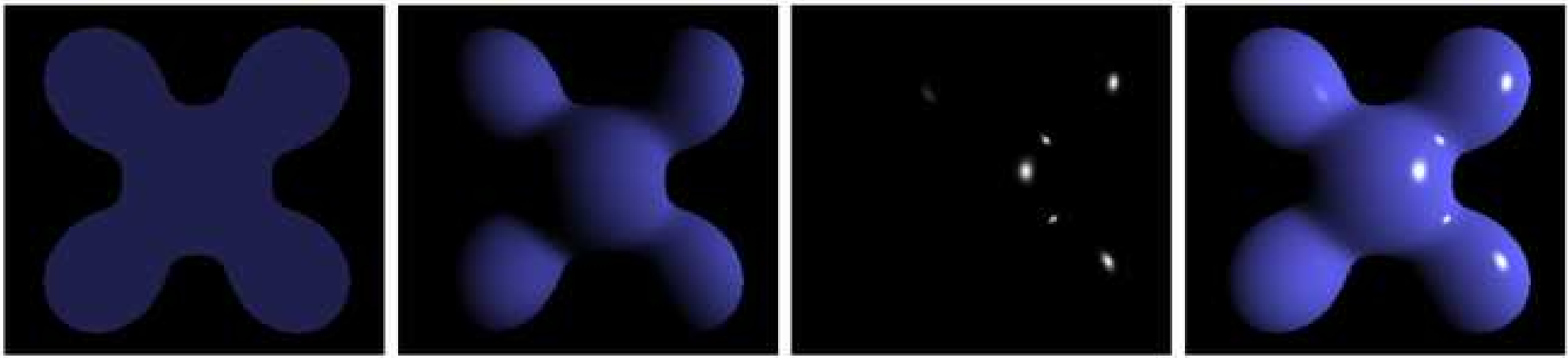


- **local** illumination: direct lighting
- **global** illumination: direct lighting + indirect lighting

Phong Illumination Model

an empirical model of the **local** illumination of points on a surface

- efficient, simple, widely used in real-time rendering
- lights and materials have three separate color components
  - ambient, diffuse, specular
- the Phong Equation calculates and sums their interaction



Phong Equation

$$I = \sum_{\text{lights}} (I_d + I_s) + I_a \tag{1}$$

for each light

- I_d approximates **diffuse** reflection (related to surface roughness)
- I_s approximates **specular** reflection (related to surface shininess)

globally

- I_a **ambient** light (light scattered around the scene)
  - calculated without correlation to the lights properties
  - ...but approximates all the indirect lighting!
    - ★ the biggest **cheat** in real-time rendering

Ambient Occlusion

- a "fake" global illumination technique
  - the ambient light is scaled by an **accessibility factor** derived from the scene geometry
  - we can enhance the Phong equation through this factor:

$$I = \sum_{\text{lights}} (I_d + I_s) + I_a * AO \tag{2}$$

Ambient Occlusion

- a "fake" global illumination technique
  - the ambient light is scaled by an **accessibility factor** derived from the scene geometry
  - we can enhance the Phong equation through this factor:

$$I = \sum_{\text{lights}} (I_d + I_s) + I_a * AO \tag{2}$$

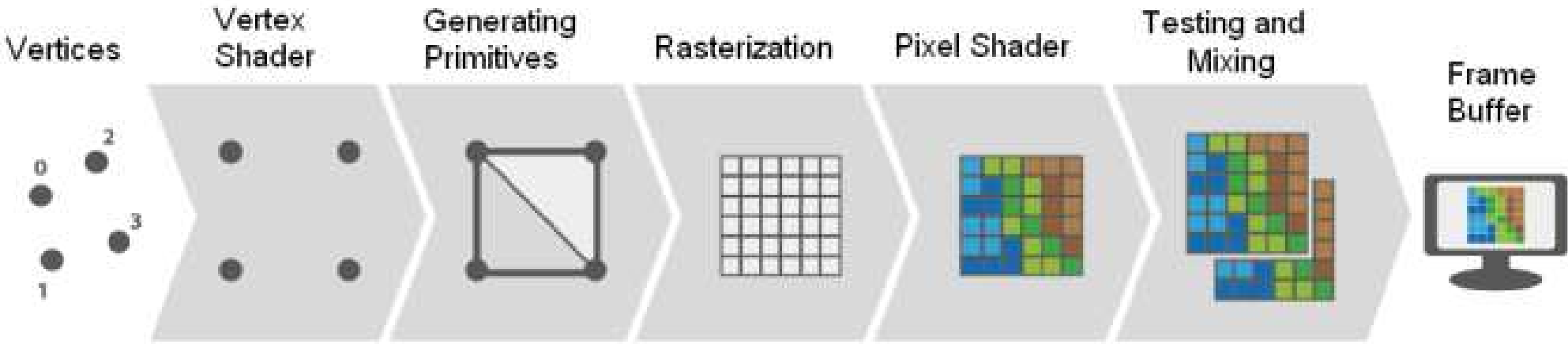
- different techniques calculate the accessibility factor in different ways
  - the basic idea is that concave/occluded areas receive less light

Ambient Occlusion

- real indirect light depends on the light properties of the scene
  - ambient occlusion doesn't
- result: soft shadows, non directional, static
  - anyway, visual perception of volumes is greatly enhanced

Real time rendering implementation

- contemporary real time rendering is based on rasterization
- the **graphics pipeline** implements the process



- **rasterization** is the process of computing the mapping from scene geometry to pixels
- **shading** is the particular way we compute the color of those pixels, depending on some lighting model

Fixed VS Programmable graphics pipeline

- until some years ago, graphics hardware implemented a (configurable) **fixed function pipeline**
  - a specific lighting model (Phong) was hardcoded
- today, we have **programmable graphics hardware**
  - programmers can define many stages of the rendering pipeline through shaders...
  - ...implementing custom lighting models and special effects
- **shaders** are programs written in domain specific languages (shading languages)
  - are executed directly on the GPU and exploit **parallelism**

OpenGL

- We access the graphics hardware through a software interface
  - basically, OpenGL or DirectX - here, we use OpenGL
- **OpenGL** is an API specification for rendering graphics
  - hardware vendors implement the spec in the video card drivers
  - we access the implementation through software libraries
- **C/C++** are the common choice for OpenGL programming
  - but there are OpenGL bindings for many programming languages

Goal

- evaluating and experimenting the applicability of the approach used in **Disk Based Screen Space Ambient Occlusion** to approximate **Global Illumination** in **real-time rendering**
  - Disk Based SSAO is an ambient occlusion technique developed by Emilio Spatola last year in this lab

Goal

- evaluating and experimenting the applicability of the approach used in **Disk Based Screen Space Ambient Occlusion** to approximate **Global Illumination** in **real-time rendering**
  - Disk Based SSAO is an ambient occlusion technique developed by Emilio Spatola last year in this lab
- ...Ok!

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering

4 Disk Based SSAO strikes back

5 Screen Space Global Illumination

6 Conclusions

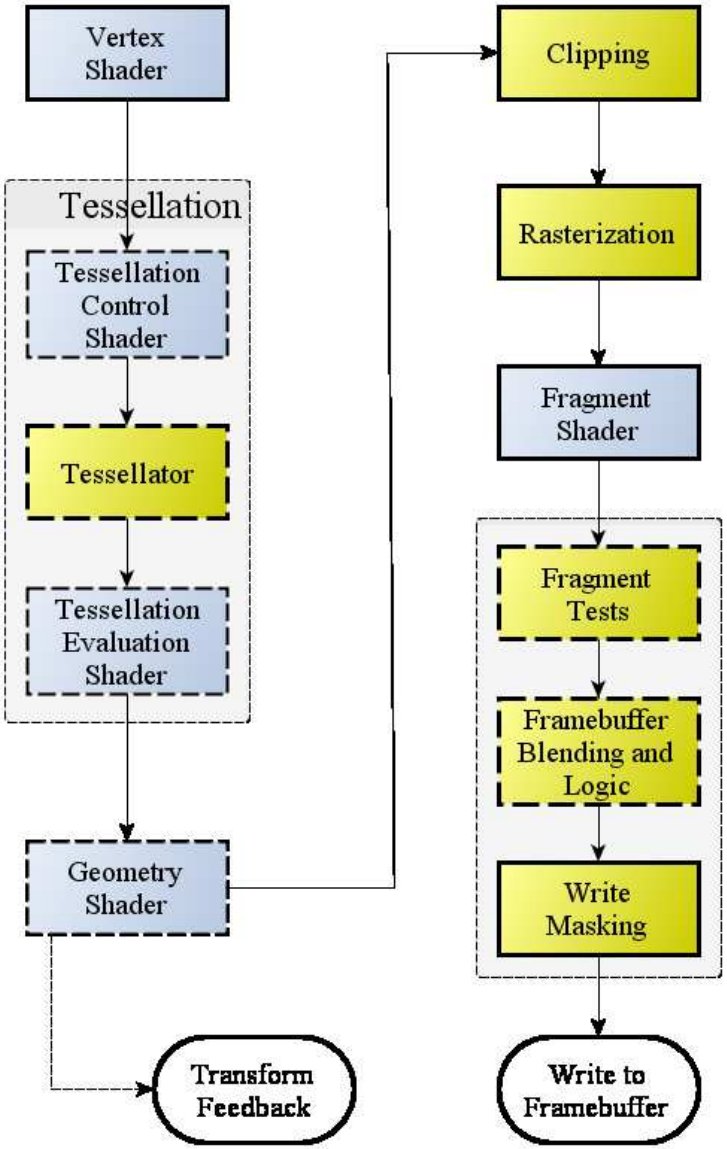
Modern OpenGL

- modern OpenGL is heavily **shader based**
  - we are going to use **GLSL**, the vendor-agnostic OpenGL Shading Language
  - Nvidia **CG** is another popular shading language that can be used with OpenGL

Modern OpenGL

- modern OpenGL is heavily **shader based**
  - we are going to use **GLSL**, the vendor-agnostic OpenGL Shading Language
  - Nvidia **CG** is another popular shading language that can be used with OpenGL
- the fixed function pipeline has been completely **deprecated**
  - nowadays video cards don't have specific hardware that implements it
    - ★ they have some built-in shaders that mimic it, to provide backwards compatibility
  - more **flexibility** and **efficiency**
    - ★ but getting started is a little harder

OpenGL Rendering Pipeline overview



- the blue boxes are programmable stages
- vertex, geometry, fragment shaders
- tesselation and compute shaders are recent additions
  - let's forget about them

OpenGL ES & WebGL

- OpenGL subsets suitable for usage in
  - **embedded devices** (smartphones, Raspberry PI...)
  - **web browsers**, through Javascript
- follow the same shader based approach
  - another reason to forget about "old" OpenGL

GLSL

The OpenGL Shading Language

- C-like
  - some limitations: no pointers, no recursion
- graphics programming friendly
  - data types and functions commonly used in computer graphics are built-in
- the language is simple
  - but **communication** with the host application can be tricky

Supporting libraries

- C++, OpenGL, GLSL... are we ready to rock?
- **not really**!
  - OpenGL is purely concerned with rendering

Supporting libraries

- C++, OpenGL, GLSL... are we ready to rock?
- **not really**!
  - OpenGL is purely concerned with rendering
- to be productive in reasonable time more **supporting libraries** are needed
  - ...and **choosing** which ones can be time consuming

Supporting libraries

- C++, OpenGL, GLSL... are we ready to rock?
- **not really**!
  - OpenGL is purely concerned with rendering
- to be productive in reasonable time more **supporting libraries** are needed
  - ...and **choosing** which ones can be time consuming
- I tried to **pick** libraries that are
  - widely used and actively developed
  - portable
  - lightweight and unobtrusive
- ...while I **avoided** libraries who could influence rendering control
  - scene graph handling
  - rendering loop ownership

Supporting libraries: my picks

- OpenGL extensions handling: **GLEW**
- OpenGL context, windowing and input handling: **GLFW**
- 3D math: **GLM** (OpenGL Mathematics)
- 3D asset loading: **Assimp**
- image loading and handling: DevIL, FreeImage, ImageMagick, GLI
- GUI overlay: **AntTweakBar**

- want more details?
  - http://blog.duskzone.it/2013/03/04/learning-opengl-getting-started-and-picking-the-right-libraries/

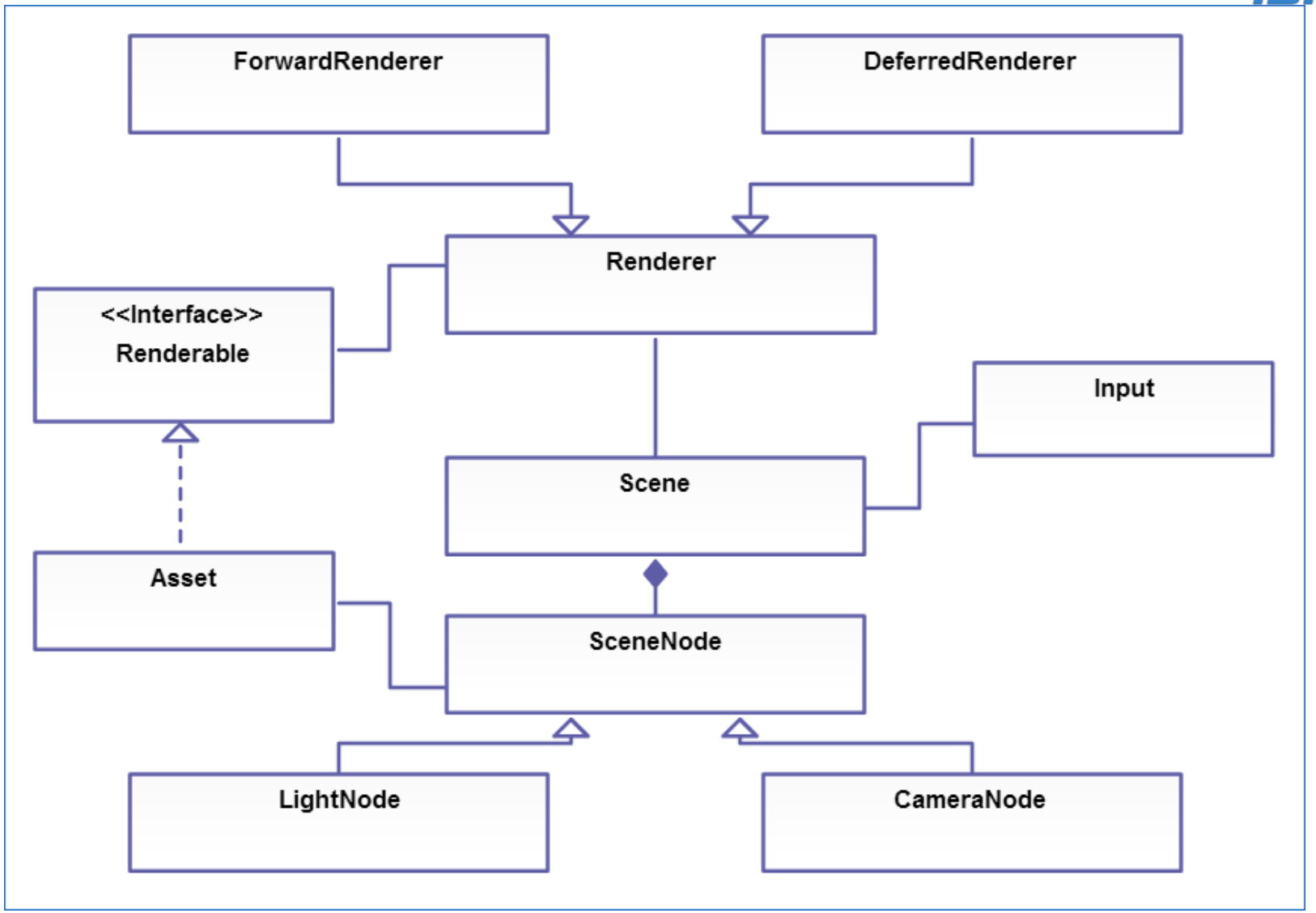
Sandbox architecture

- not the cleanest design possible
- features added when needed
- heavy experimentation and refactoring on the go
  - good reasons to not show lots of code today!

Sandbox architecture

- not the cleanest design possible
- features added when needed
- heavy experimentation and refactoring on the go
  - good reasons to not show lots of code today!
- every week, Ugo reminds me
  - "Don't try writing a 3D engine!"
- fortunately the word **sandbox** came to my mind :)

Sandbox architecture



Sandbox architecture

- basically:
  - we have a **Renderer**, a **Scene** and user **Input**

Sandbox architecture

- basically:
  - we have a **Renderer**, a **Scene** and user **Input**
- a Scene is made up of a **SceneNode** tree
  - each SceneNode carries a transformation matrix (and more)

Sandbox architecture

- basically:
  - we have a **Renderer**, a **Scene** and user **Input**
- a Scene is made up of a **SceneNode** tree
  - each SceneNode carries a transformation matrix (and more)
- user Input/time alter the Scene
  - for example, moving a SceneNode by an offset vector, or rotating the camera
  - special SceneNodes handle Lights and Cameras

Sandbox architecture

- basically:
  - we have a **Renderer**, a **Scene** and user **Input**
- a Scene is made up of a **SceneNode** tree
  - each SceneNode carries a transformation matrix (and more)
- user Input/time alter the Scene
  - for example, moving a SceneNode by an offset vector, or rotating the camera
  - special SceneNodes handle Lights and Cameras
- a **Renderable** is an object that can be rendered by a Renderer
- **Assets** are loaded 3D models
  - implementing the Renderable interface

The Render Loop

So, for each frame

- the Scene is updated (according to user Input)
- the instanced Renderer navigates the Scene tree and renders the attached Renderables

```
do {
    double currentTime = glfwGetTime();
    double deltaTime = currentTime - lastTime;
    lastTime = currentTime;
    int msec = deltaTime * 1000;

    scene.update(&input, msec);
    renderer.render(&scene);
    gui.render();

    glfwSwapBuffers();
} while( closeRequested() );
```

Scene loading

- the **Assimp** library parses the **model files**
- we get an array of **Meshes** and a tree defining their hierarchy
  - a Mesh, for Assimp, represents a piece of geometry with a single material
    - **geometry data** consists of vertices, UV coords, normals, tangents...
    - a **material** is defined by a set of attributes and textures
- we upload to the GPU memory all the data at startup
- we save the hierarchical structure
  - attaching to each node references to its data

Scene rendering

For each frame, we walk the Scene hierarchical structure and render each node

- an oversimplification:
  - OpenGL calls in the **host application** just set some handles to define **what** to draw
  - **shaders** define **how** to draw it

Scene rendering

For each node:

- in the application code:
  - we enable a **VAO** (Vertex Array Object)
    - ★ a VAO encapsulates references to all the geometric attributes of a single mesh
  - we set the basic material properties through a **UBO** (Uniform Buffer Object)
    - ★ diffuse/ambient/specular/emissive color, shininess
  - we activate its **textures**
    - ★ diffuse/normal map/specular map/opacity map
  - we call `glDrawElements`
- in the currently bound shader
  - all the incoming data is used to do draw to an output buffer

Shaders must-know

- how data flows
  - between application and shaders
    - ★ vertex attributes binding
    - ★ uniform data
    - ★ texture sampling
    - ★ output location
  - between different shader stages
    - ★ interface blocks
    - ★ interpolation

Shaders must-know

- how many times shader stages are executed
- a simplified example:
  - we define a vertex buffer of 6 vertex (two triangles) in the application and draw it
    - ★ we set as attributes the vertex position and color
  - the vertex shader is executed 6 times, once for each vertex
    - ★ we can access the attributes of a single vertex
  - the geometry shader is executed 2 times, once for triangle
    - ★ we can access the attributes of the three vertices of the current triangle
  - the fragment shader is executed for each pixel the triangles cover
    - ★ usually, input values coming from the previous shader are interpolated
    - ★ for a fragment in the middle of a red vertex and a blue vertex, we get purple as color

Shaders must-know

- computations must be done in the right place, e.g.:
  - transforming a light position to camera space should be done in the application and not in the VS
    - ★ if we do it in the VS we do the same computation for each vertex (potentially thousands of times)
    - ★ even worse if we do it in the FS (typically, we have more fragments than vertices)
- the data evaluated in the application is sent to shaders as **uniform**
  - global, readonly, accessible from all the stages

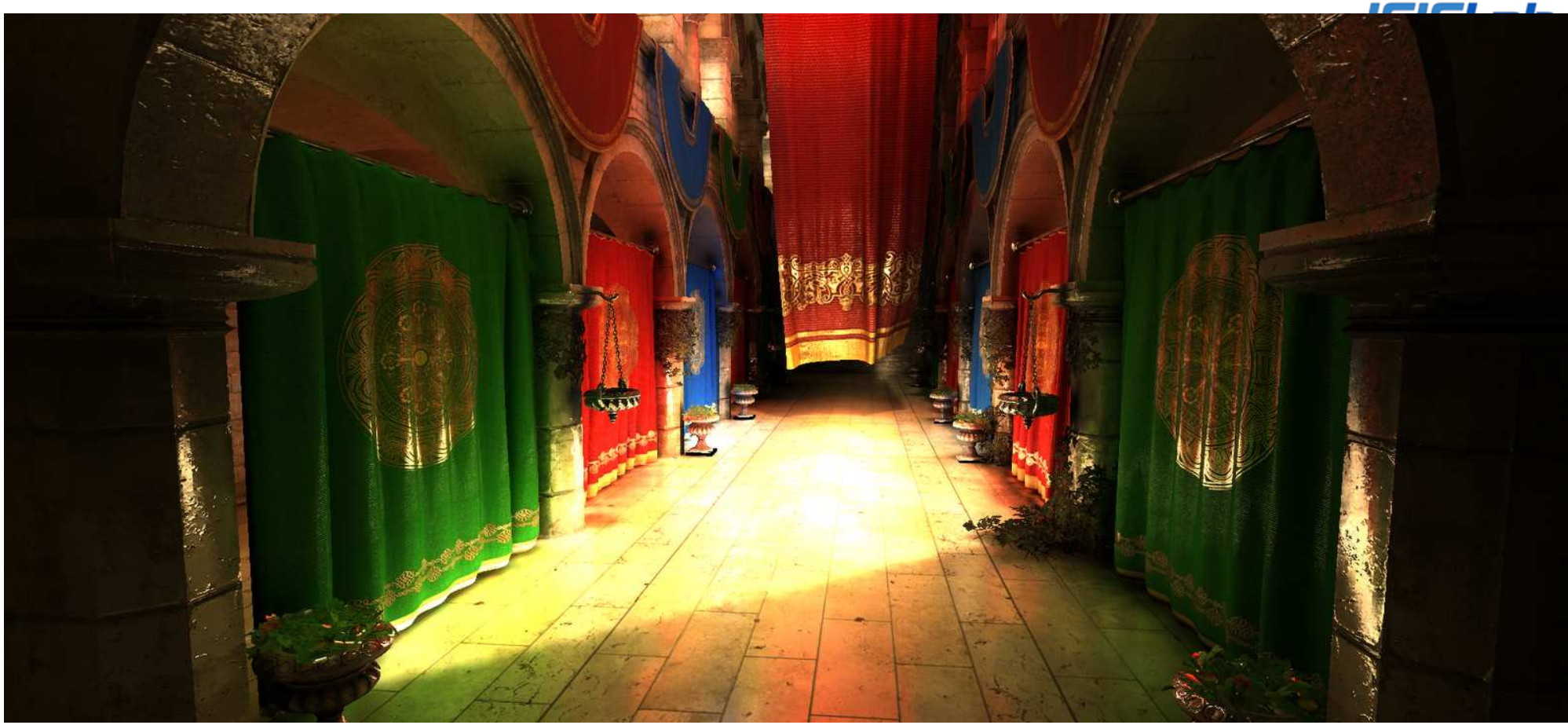
The Sponza Atrium

- located in Dubrovnik, Croatia
- spawned two models popular in CG
  - 2002: by Marko Dabrovic
    - ★ simple for nowadays standards
  - 2010: by Crytek
    - ★ featuring many more details and normal maps, specular maps, opacity maps

The Sponza Atrium

- located in Dubrovnik, Croatia
- spawned two models popular in CG
  - 2002: by Marko Dabrovic
    - ★ simple for nowadays standards
  - 2010: by Crytek
    - ★ featuring many more details and normal maps, specular maps, opacity maps
- our sandbox has been built to fully support the model by Crytek
  - but can of course handle every model with similar (or less) features that can be loaded by Assimp

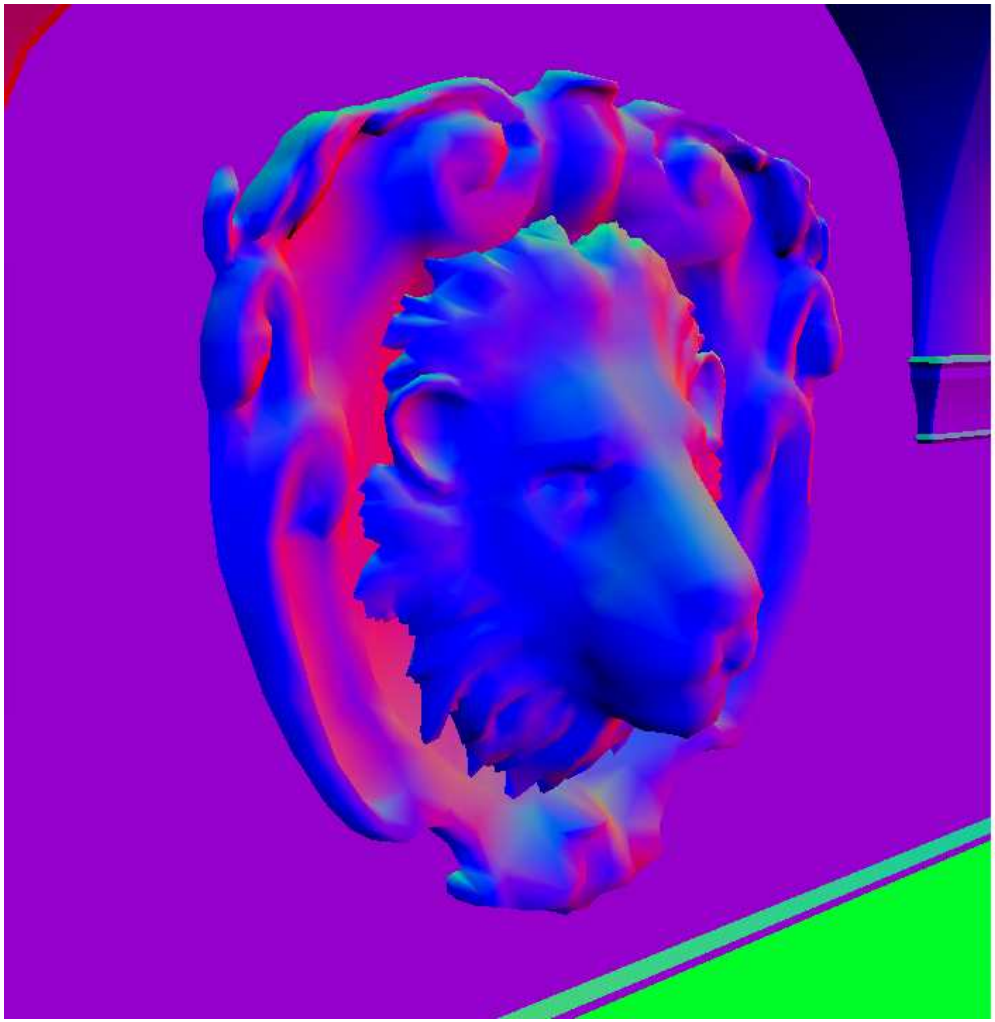
The Sponza Atrium

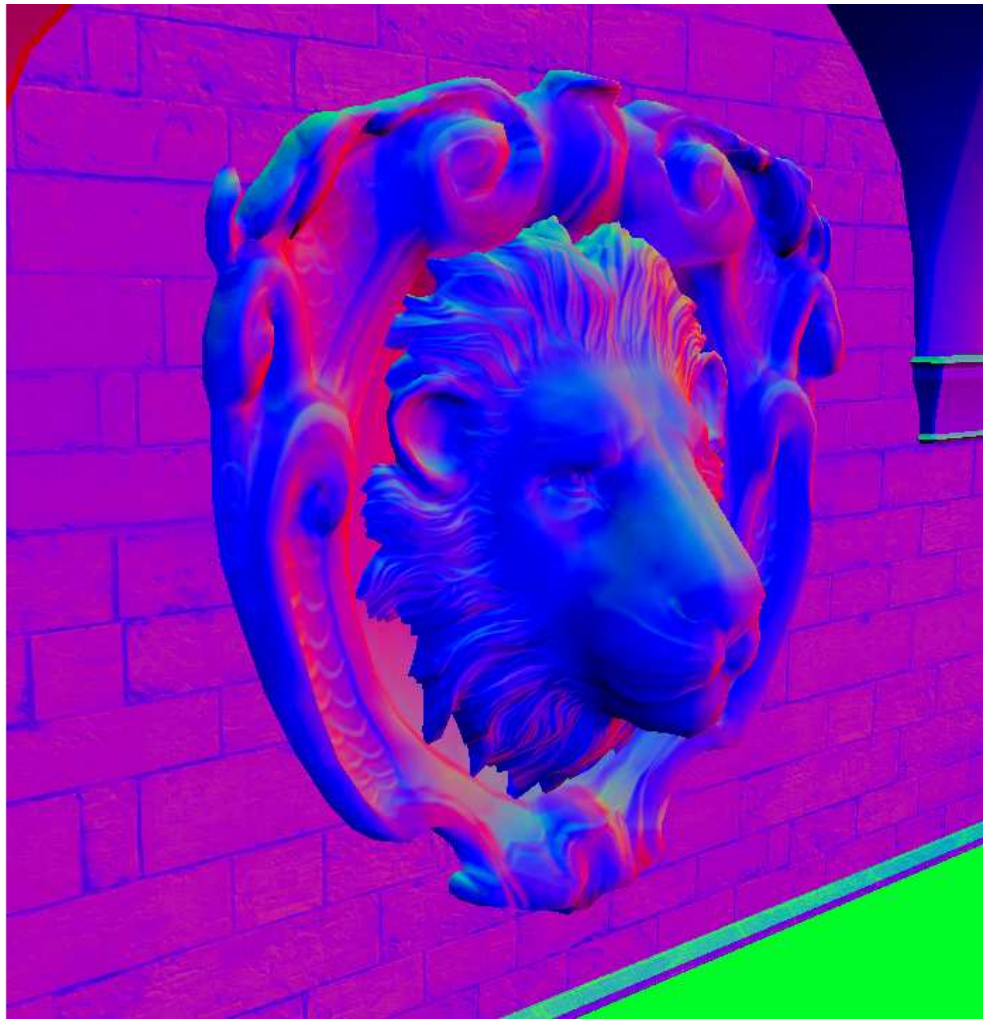


the Crytek Sponza Atrium is probably the most used scene in Global Illumination demos

- our ultimate target!

The Sponza Atrium: normal mapping





The Sponza Atrium: alpha masking

The Sponza Atrium: specular maps

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering

4 Disk Based SSAO strikes back

5 Screen Space Global Illumination

6 Conclusions

Forward Rendering vs Deferred Rendering

- forward rendering is... rendering
  - one pass:
    - ★ each object in the scene, with no particular order, is rendered to the framebuffer calculating its surface color (based on its material and all the lights that affect it)
    - ★ the depth buffer is used to decide if a pixel must be drawn over or not

Forward Rendering vs Deferred Rendering

- forward rendering is... rendering
  - one pass:
    - ★ each object in the scene, with no particular order, is rendered to the framebuffer calculating its surface color (based on its material and all the lights that affect it)
    - ★ the depth buffer is used to decide if a pixel must be drawn over or not
- deferred rendering **delays the shading** to a second pass
  - two passes:
    - ★ 1) each object in the scene, with no particular order, is rendered to a special buffer (the GBuffer) saving the information needed for shading
    - ★ 2) the shading is done taking info from the GBuffer and writing to the framebuffer

Deferred Rendering pros

What's the point?

- **decoupling** the geometry processing from the shading calculation
  - shading is the most **computationally heavy** part of rendering

Deferred Rendering pros

What's the point?

- **decoupling** the geometry processing from the shading calculation
  - shading is the most **computationally heavy** part of rendering

forward rendering:

- $O(geometriesSurfacePixels * lights)$

deferred rendering:

- $O(screenPixels * lights)$

Deferred Rendering pros

What's the point?

- **decoupling** the geometry processing from the shading calculation
  - shading is the most **computationally heavy** part of rendering

forward rendering:

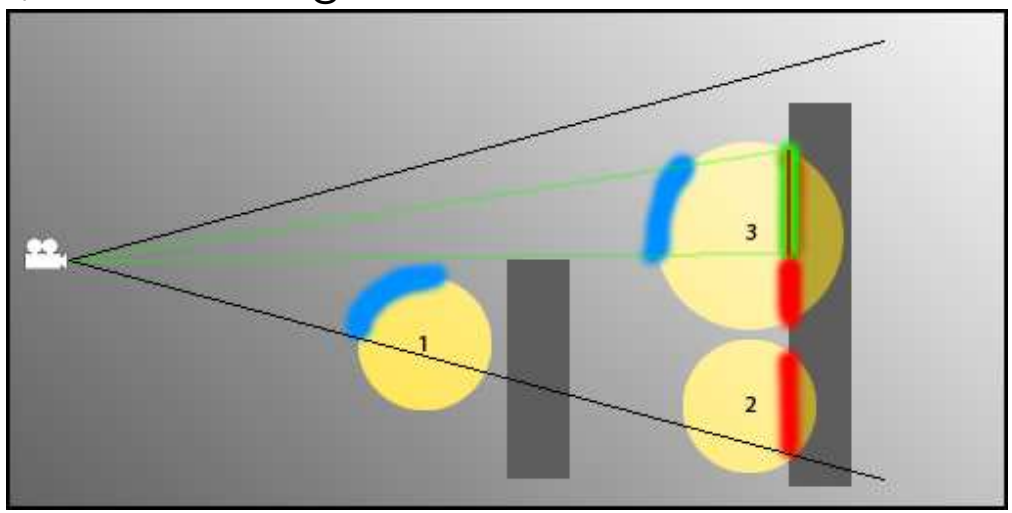
- $O(geometriesSurfacePixels * lights)$

deferred rendering:

- $O(screenPixels * lights)$
- geometry is still rendered to the GBuffer
- but only visible pixels of surfaces are shaded
  - zero light calculation overdraw

Deferred Rendering optimizations

- $O(screenPixels * lights)$ is an upper bound
  - much better can be done with some optimizations
    - ★ light volumes, stencil culling



    - ★ only "green" pixels are shaded, and only by light 3
  - I'm currently implementing these optimizations
    - ★ they should allow using even hundreds (thousands?) of (small) dynamic lights

Forward Rendering in my sandbox

- uses a single shader program
  - forward (.vert + .frag)
- we enable it and draw all the objects to the default framebuffer
- all the lights data are accessible to the shader
  - loop in the fragment shader: for each light, shade pixel

Deferred Rendering in my sandbox

- two shader programs
  - deferred (.vert + .geom + .frag)
  - deferred_lightpass (.vert + .frag)
- geometry pass:
  - we enable "deferred" and draw all the objects to the GBuffer with MRT
- lighting pass:
  - we switch to the default framebuffer
  - we enable "deferred_lightpass" and a draw a fullscreen quad
    - ★ doing all the shading

Work in progress

- three shader programs
  - deferred (.vert + .geom + .frag)
  - deferred_lightpass (.vert + .frag)
  - deferred_lightpass_pointlight (.vert + .frag)
- geometry pass:
  - we enable "deferred" and draw all the objects to the GBuffer with MRT
- lighting pass:
  - we switch to the default framebuffer and enable **additive blending**
  - we use **light volumes** and **stencil culling** to limit the shading calculations
    - ★ we enable "deferred_lightpass_pointlight" and draw a **sphere** for each **pointlight**
  - we enable "deferred_lightpass" and a draw a **fullscreen quad**
    - ★ global operations (ambient lighting, ambient occlusion)

Work in progress

- three shader programs
  - deferred (.vert + .geom + .frag)
  - deferred_lightpass (.vert + .frag)
  - deferred_lightpass_pointlight (.vert + .frag)
- geometry pass:
  - we enable "deferred" and draw all the objects to the GBuffer with MRT
- lighting pass:
  - we switch to the default framebuffer and enable **additive blending**
  - we use **light volumes** and **stencil culling** to limit the shading calculations
    - ★ we enable "deferred_lightpass_pointlight" and draw a **sphere** for each **pointlight**
  - we enable "deferred_lightpass" and a draw a **fullscreen quad**
    - ★ global operations (ambient lighting, ambient occlusion)
- additional shaders for directional/point lights (drawing full screen quads and cones) can be added

GBuffer, FBO, MRT

- the GBuffer is implemented with a **FBO** (Framebuffer Object)
  - user defined framebuffer
  - we can render to the FBO
    - ★ as we render to the default framebuffer (what we see on screen)
  - we can attach different textures to the FBO
    - ★ and we can render to them **at once**
- the **MRT** (Multiple Render Targets) feature allows this processing
  - its implementation in recent GPUs made deferred rendering popular

Writing to the GBuffer

- we create some textures and attach them to the FBO "**color attachment points**"
- we bind the FBO for writing
- we setup **multiple outputs** in the fragment shader

```
layout(location = 0) out vec4 positionTexData;
layout(location = 1) out vec4 diffuseTexData;
layout(location = 2) out vec4 normalTexData;
layout(location = 3) out vec4 specularTexData;

//...
specularTexData  = vec4(specularColor.rgb, shininess);
```

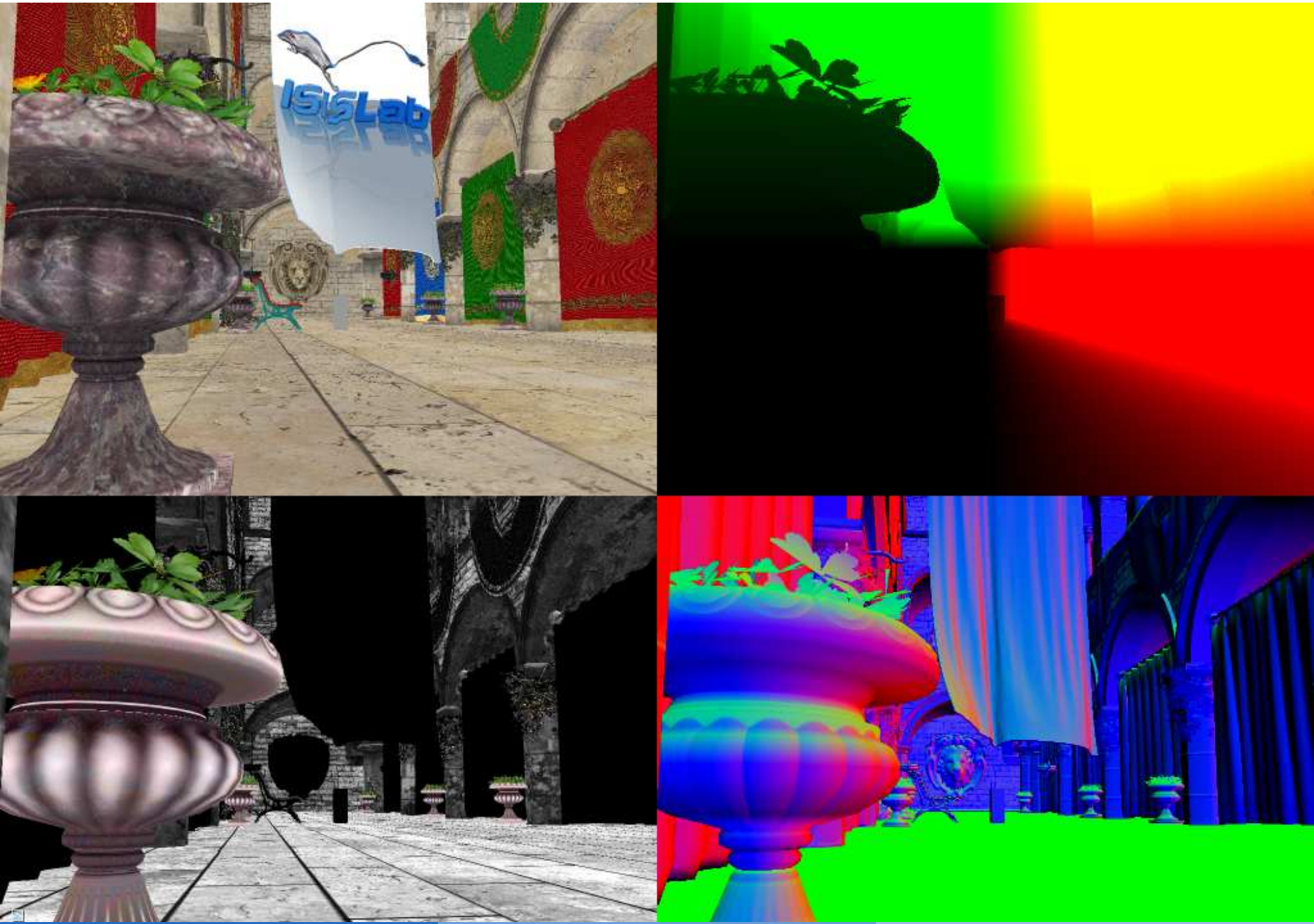
- the values we write into these variables in the fragment shader go to the attached textures

Reading from the GBuffer

in the second pass

- we activate the FBO textures
- we sample them from the shaders as with every other texture and get the data back

```
uniform sampler2D positionSampler;
uniform sampler2D diffuseSampler;
uniform sampler2D normalSampler;
uniform sampler2D specularSampler;
//...
vec4 specularColor = texture(specularSampler, coords);
float shininess = specularColor.a;
specularColor.a = 1.0;
```


ISISLab

GBuffer layout

The GBuffer layout needs special care

- bandwidth and memory requirements
- many optimizations possible and widely applied
  - using lower numeric precision when possible
  - encoding normals (storing only x, y and calculating z)
  - deriving 3D position from 2D position + depth buffer
  - passing monochromatic specular info (but Crytek Sponza has some colored specular maps)

GBuffer layout

The GBuffer layout needs special care

- bandwidth and memory requirements
- many optimizations possible and widely applied
  - using lower numeric precision when possible
  - encoding normals (storing only x, y and calculating z)
  - deriving 3D position from 2D position + depth buffer
  - passing monochromatic specular info (but Crytek Sponza has some colored specular maps)
- I haven't applied any optimization yet
  - removing the position buffer is #1 on my TODO list

Deferred Rendering issues

- So is deferred rendering better?
- Can we forget about forward rendering?

Deferred Rendering issues

- So is deferred rendering better?
- Can we forget about forward rendering?
  - no
  - at least, not yet
    - ★ some issues to sort out

Deferred Rendering issues: alpha blending

Alpha blending gets difficult

- with forward rendering, you just draw transparent objects back to front, blending
- with deferred rendering, you miss information about the objects "behind" the closest one

Deferred Rendering issues: alpha blending

Alpha blending gets difficult

- with forward rendering, you just draw transparent objects back to front, blending
- with deferred rendering, you miss information about the objects "behind" the closest one

Possible solutions:

- draw only the transparent objects with a forward rendering pass
- use some more sophistcated techniques
  - "Order Independent Transparency with Dual **Depth Peeling**"
    - ★ Louis Bavoil, Kevin Myers (NVIDIA)

Deferred Rendering issues: alpha blending

Alpha blending gets difficult

- with forward rendering, you just draw transparent objects back to front, blending
- with deferred rendering, you miss information about the objects "behind" the closest one

Possible solutions:

- draw only the transparent objects with a forward rendering pass
- use some more sophistcated techniques
  - "Order Independent Transparency with Dual **Depth Peeling**"
    - ★ Louis Bavoil, Kevin Myers (NVIDIA)

Or you forget about supporting alpha blending, as I did!

- anyway: Crytek Sponza is ok with just alpha masking

Deferred Rendering issues: materials shading

Often, with forward rendering, different kind of materials are shaded with different shaders

- implementing illumination models better approximating the **material physical properties**
- e.g., we enable the "shinyMetal" shader and draw all the metallic objects in the scene

Deferred Rendering issues: materials shading

Often, with forward rendering, different kind of materials are shaded with different shaders

- implementing illumination models better approximating the **material physical properties**
- e.g., we enable the "shinyMetal" shader and draw all the metallic objects in the scene

But with deferred rendering, during the light-pass, we only have the GBuffer texture data

- we shade a pixel but don't know what object it belongs

Deferred Rendering issues: materials shading

A possible solution is writing into the GBuffer a material id and then **branching** in the light-pass shader

- due to how GPUs work, branching could kill the performances
- ...but this should be becoming less important with current hardware
  - **dynamic branching** and **pixel local coherence**

Deferred Rendering issues: materials shading

A possible solution is writing into the GBuffer a material id and then **branching** in the light-pass shader

- due to how GPUs work, branching could kill the performances
- ...but this should be becoming less important with current hardware
  - **dynamic branching** and **pixel local coherence**

Or you forget about supporting different illumination models and stick to Phong, as I did!

- anyway: Crytek Sponza doesn't specify extra-Phong material info

Deferred Rendering issues: hardware MSAA

GPUs do hardware accelerated MultiSample Anti-Aliasing

- with forward rendering, it just needs to be enabled
- **not so easy** with deferred rendering
  - not so easy even explaining why
  - let's say that MRT/GBuffer and MSAA don't play nice together

Deferred Rendering issues: hardware MSAA

GPUs do hardware accelerated MultiSample Anti-Aliasing

- with forward rendering, it just needs to be enabled
- **not so easy** with deferred rendering
  - not so easy even explaining why
  - let's say that MRT/GBuffer and MSAA don't play nice together
- Solutions:
  - implementing some filtering technique in post-processing
    - ★ **FXAA** is gaining popularity, fast but causing some blurriness
  - using multisampled render targets in GBuffer and manually implementing MSAA accessing the samples (`texelFetch()`)
    - ★ it looks like nobody does this: too slow?

Deferred Rendering issues: hardware MSAA

GPUs do hardware accelerated MultiSample Anti-Aliasing

- with forward rendering, it just needs to be enabled
- **not so easy** with deferred rendering
  - not so easy even explaining why
  - let's say that MRT/GBuffer and MSAA don't play nice together
- Solutions:
  - implementing some filtering technique in post-processing
    - ★ **FXAA** is gaining popularity, fast but causing some blurriness
  - using multisampled render targets in GBuffer and manually implementing MSAA accessing the samples (`texelFetch()`)
    - ★ it looks like nobody does this: too slow?

Or you forget about anti-aliasing, as I did!

- anyway: I think I'll try to add FXAA at some point, should be easy

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering
- demo time!

4 Disk Based SSAO strikes back

5 Screen Space Global Illumination

6 Conclusions

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering

4 Disk Based SSAO strikes back

5 Screen Space Global Illumination

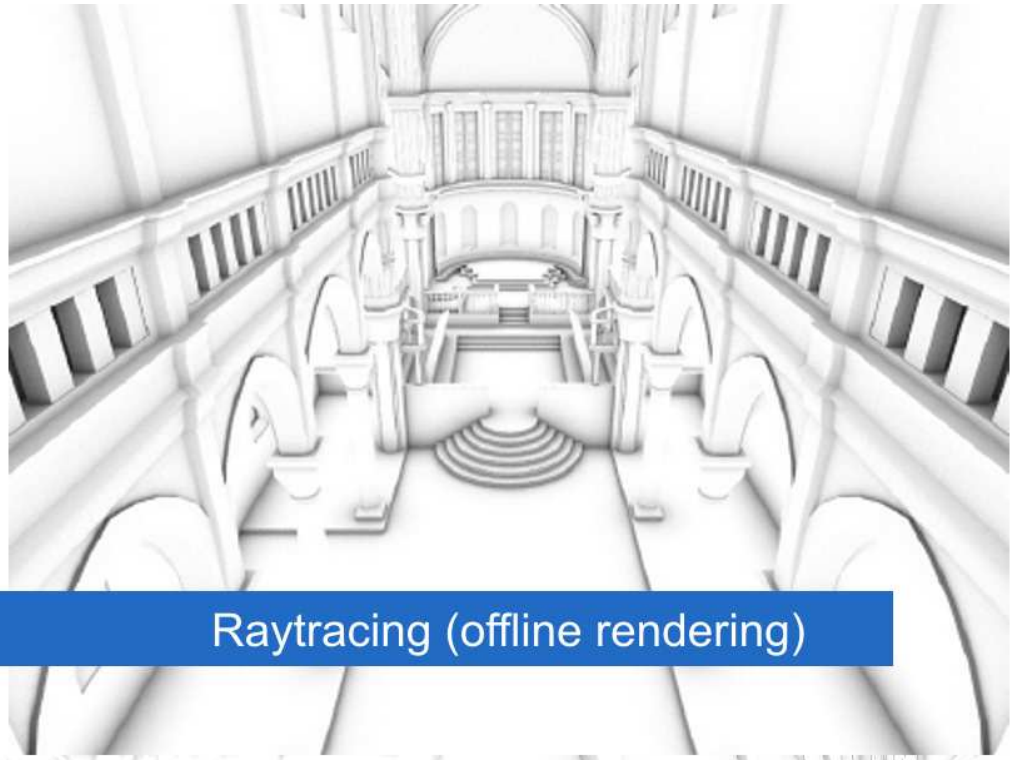
6 Conclusions

Screen Space Ambient Occlusion (SSAO)

- we already defined Ambient Occlusion (AO)
- Screen Space AO is a tecnique to efficiently approximate AO
  - it's suitable for real-time rendering
  - was used for the first time in 2007 in *Crysis*
- is a **screen space** technique
  - implemented as a fragment shader
- the AO factor of a pixel is computed sampling the **depth buffer**
  - difference between depth of the current pixel and depth of "near" ones
- no dependance on scene complexity, but relatively **low quality** result

Disk Based SSAO

- developed in 2012, here in ISISlab, by **Emilio Spatola**
- a **geometric method** based on deferred rendering
  - geometric info carried in the GBuffer is used to calculate AO
- implemented as a geometry shader and a fragment shader
  - the use of a geometry shader implies some dependecy on scene complexity
- **high quality** results, good performances



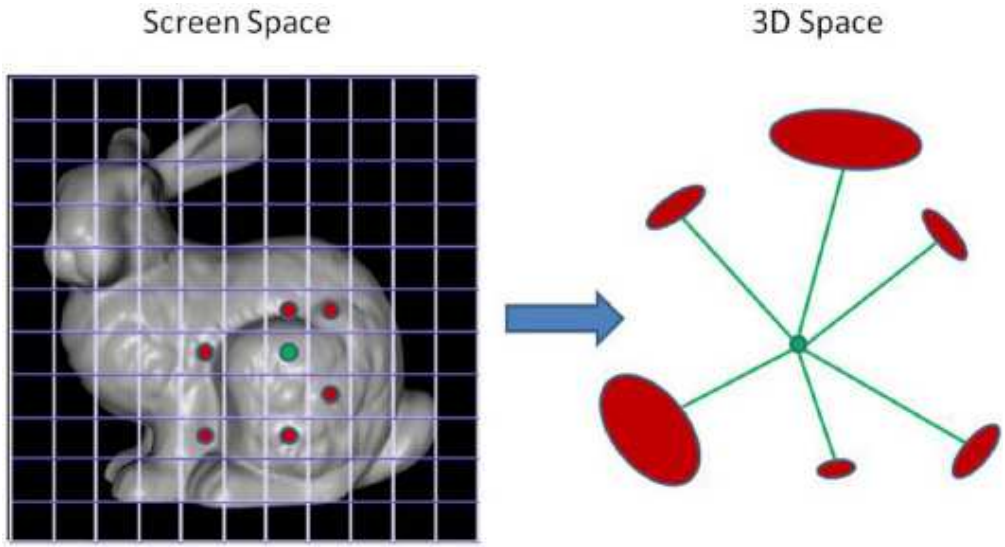
Raytracing (offline rendering)

Crytek SSAO, 73.9% SSIM, 202 fps

DSSAO, 84.7% SSIM, 133 fps

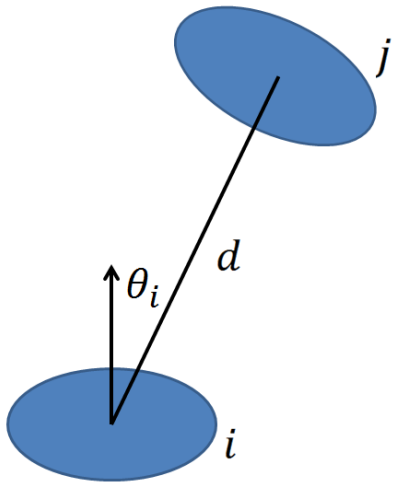
Disk Based SSAO

- when sampling "near" pixels, we go back from screen space to eye space
  - from position in the 2D image to position in the 3D scene
- so we consider the **real distance** between the two surfaces



Disk Based SSAO

- pixels are mapped to **disks**
- the AO factor is calculated according to disks areas, positions and orientations



Revisiting DSSAO

Emilio's code

- is "old style" OpenGL
- uses the Nvidia CG toolkit
- uses the 2002 Sponza model
- only shows the AO term
  - no handling of lights/materials
    - ★ black & white

Revisiting DSSAO

So, I decided to re-implement the tecnique to

- better understand how it worked
- have it working in modern OpenGL and with vendor-agnostic GLSL
- go towards GI and not only AO
  - GI demos use the 2010 Sponza model
  - we want dynamic lighting
    - ★ colors!

Revisiting DSSAO

December 2012

- "Well, I guess this is going to take a couple weeks..."

Revisiting DSSAO

December 2012

- "Well, I guess this is going to take a couple weeks..."

May 2013

- Almost done!

ISISLab

ISISLab

Why so long?

- some work to support Crytek Sponza
  - no Assimp samples using together VAOs, basic material properties and multiple textures
  - normal mapping
  - alpha masking with opacity maps

Why so long?

- some work to support Crytek Sponza
  - no Assimp samples using together VAOs, basic material properties and multiple textures
  - normal mapping
  - alpha masking with opacity maps
- some work to have "modern OpenGL" code and lighting
  - manual handling of transformations matrices, lights and material properties

Why so long?

- some work to support Crytek Sponza
  - no Assimp samples using together VAOs, basic material properties and multiple textures
  - normal mapping
  - alpha masking with opacity maps
- some work to have "modern OpenGL" code and lighting
  - manual handling of transformations matrices, lights and material properties
- some work to do little enhancements on the way
  - e.g.: a GUI to tune parameters like the number of samples at runtime

Why so long?

- some work to support Crytek Sponza
  - no Assimp samples using together VAOs, basic material properties and multiple textures
  - normal mapping
  - alpha masking with opacity maps
- some work to have "modern OpenGL" code and lighting
  - manual handling of transformations matrices, lights and material properties
- some work to do little enhancements on the way
  - e.g.: a GUI to tune parameters like the number of samples at runtime
- "Ok, stop making excuses and show us the demo"

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering

4 Disk Based SSAO strikes back
- demo time! again!

5 Screen Space Global Illumination

6 Conclusions

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering

4 Disk Based SSAO strikes back

5 Screen Space Global Illumination

6 Conclusions

It's getting late

- I'm preparing too many slides
- let's just talk about a couple things
  - **SSDO**: Screen Space Directional Occlusion
  - the idea of **Virtual Point Lights** (VPL)

Screen Space Directional Occlusion (SSDO)

- "Approximating Dynamic Global Illumination in Image Space"
  - Tobias Ritschel, Thorsten Grosch, Hans-Peter Seidel
  - I3D '09 Proceedings of the 2009 symposium on Interactive 3D graphics and games
- often referred as "the heir of SSAO"
- implemented in DirectX 11 update of CryEngine 3

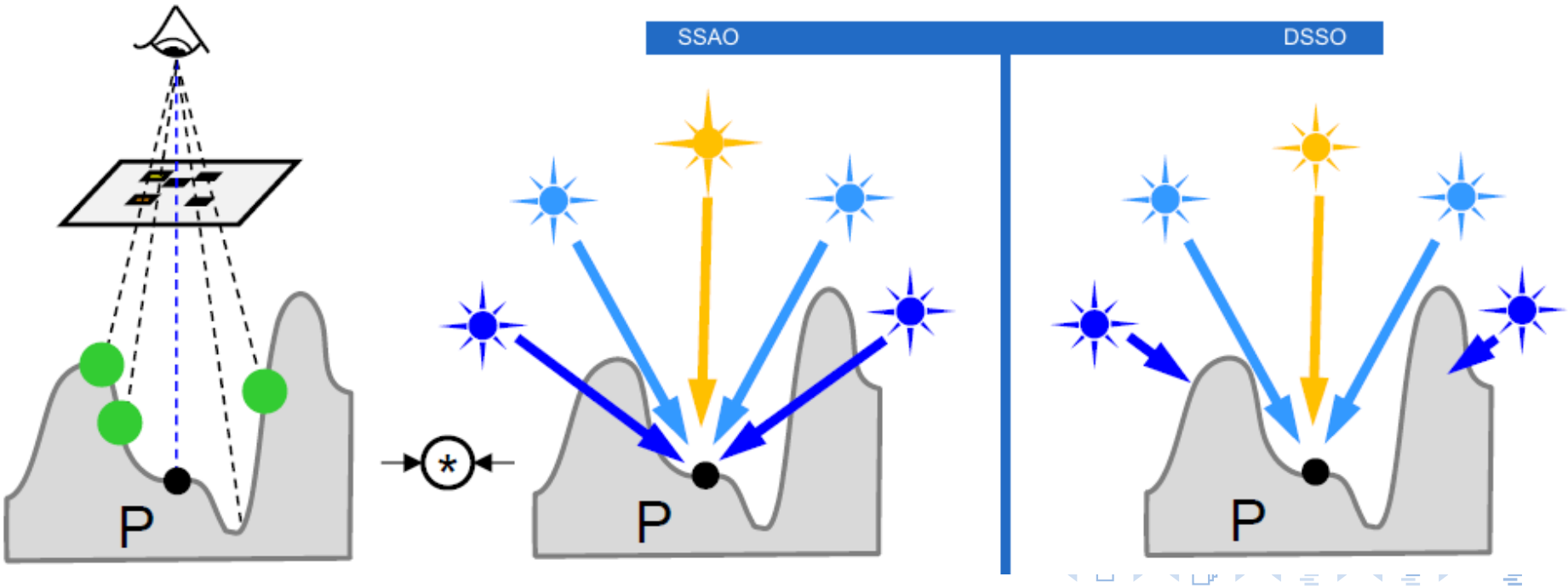
Screen Space Directional Occlusion (SSDO)

SSAO

- non directional, computing average visibility and multiplying it
- grey shadows

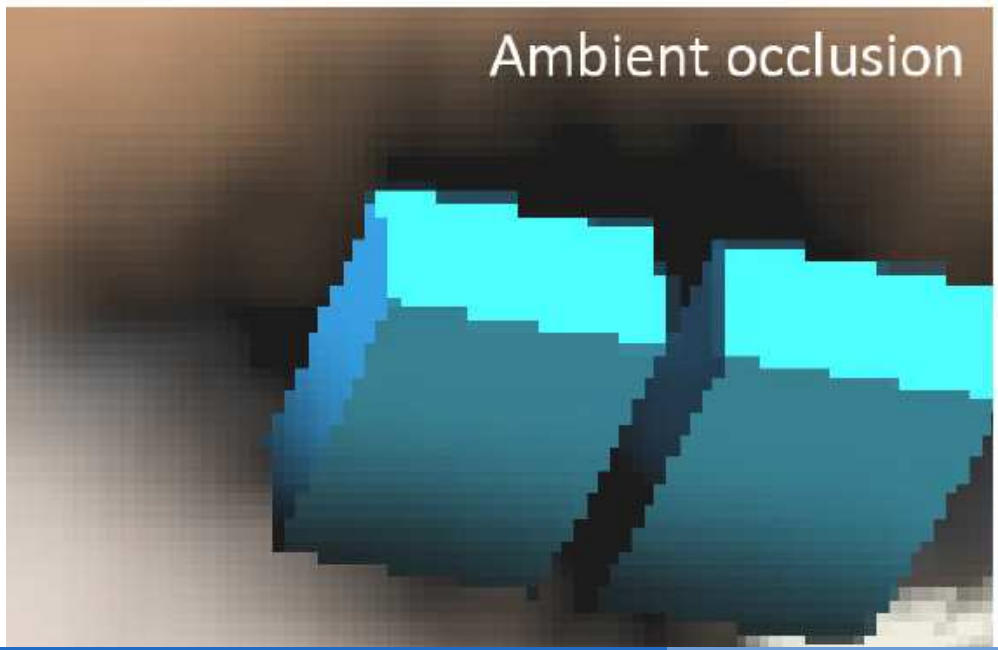
SSDO

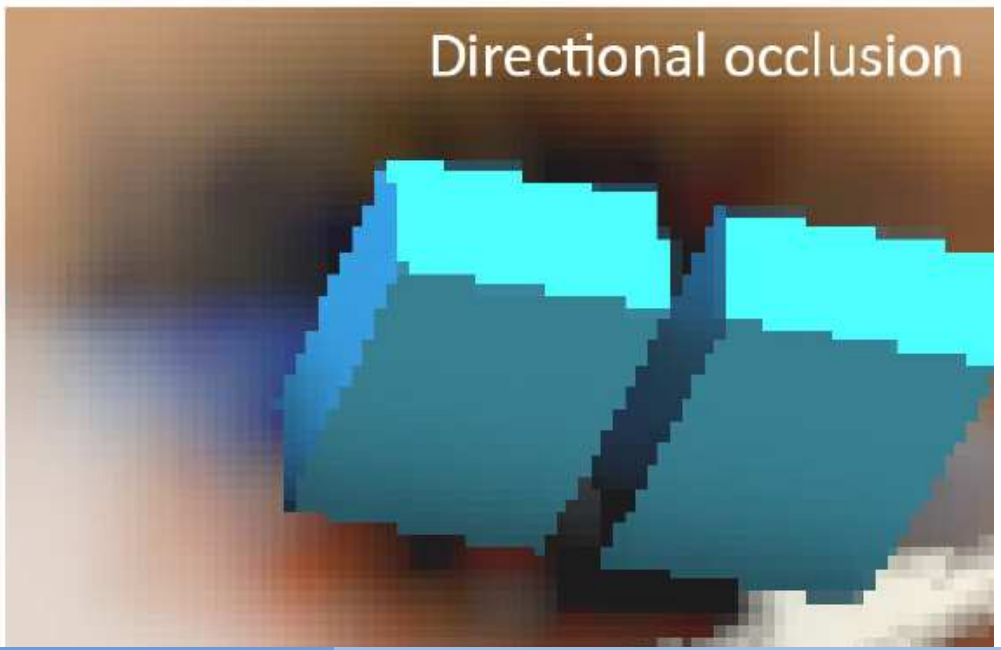
- directional, accumulates only visible illumination
- colored shadows



Ambient occlusion

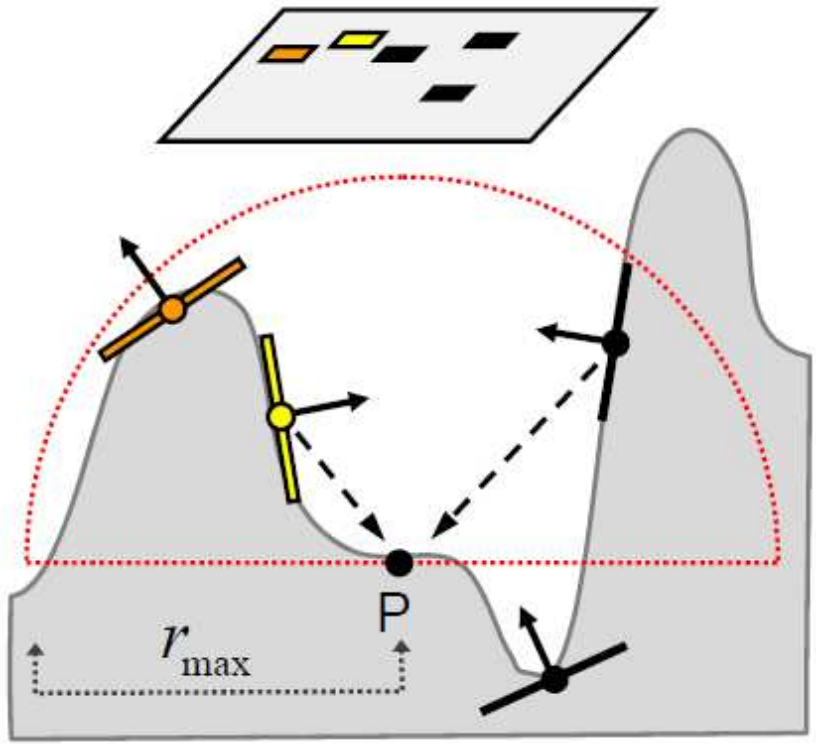
Directional occlusion

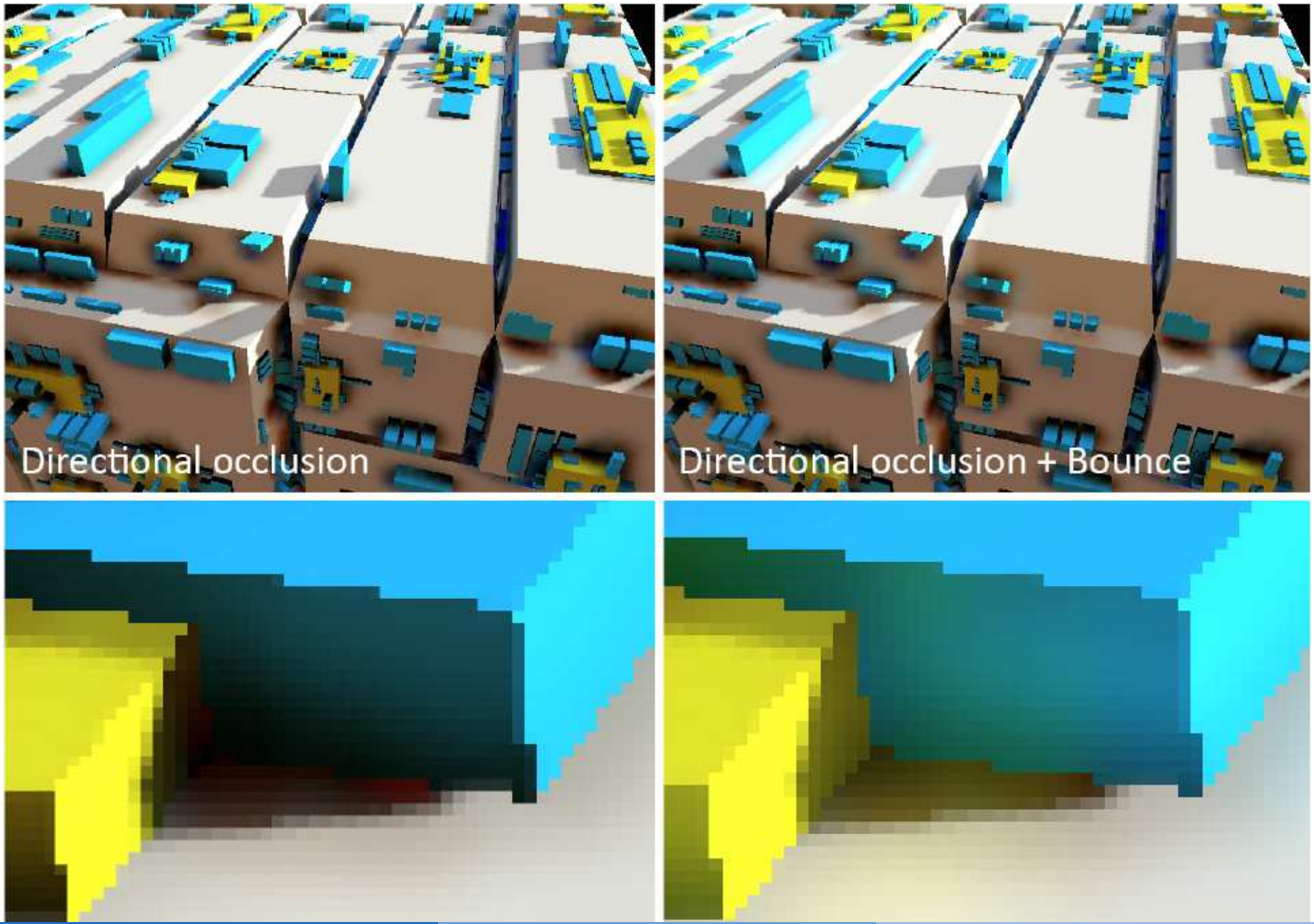

Ambient occlusion


Directional occlusion

Screen Space Directional Occlusion (SSDO)

- each DSSO sample can be considered as a small area light
  - oriented around the pixel normal
  - with radiance determined by direct light
- we can calculate one indirect bounce of light for nearby geometry
  - introducing some overhead, of course

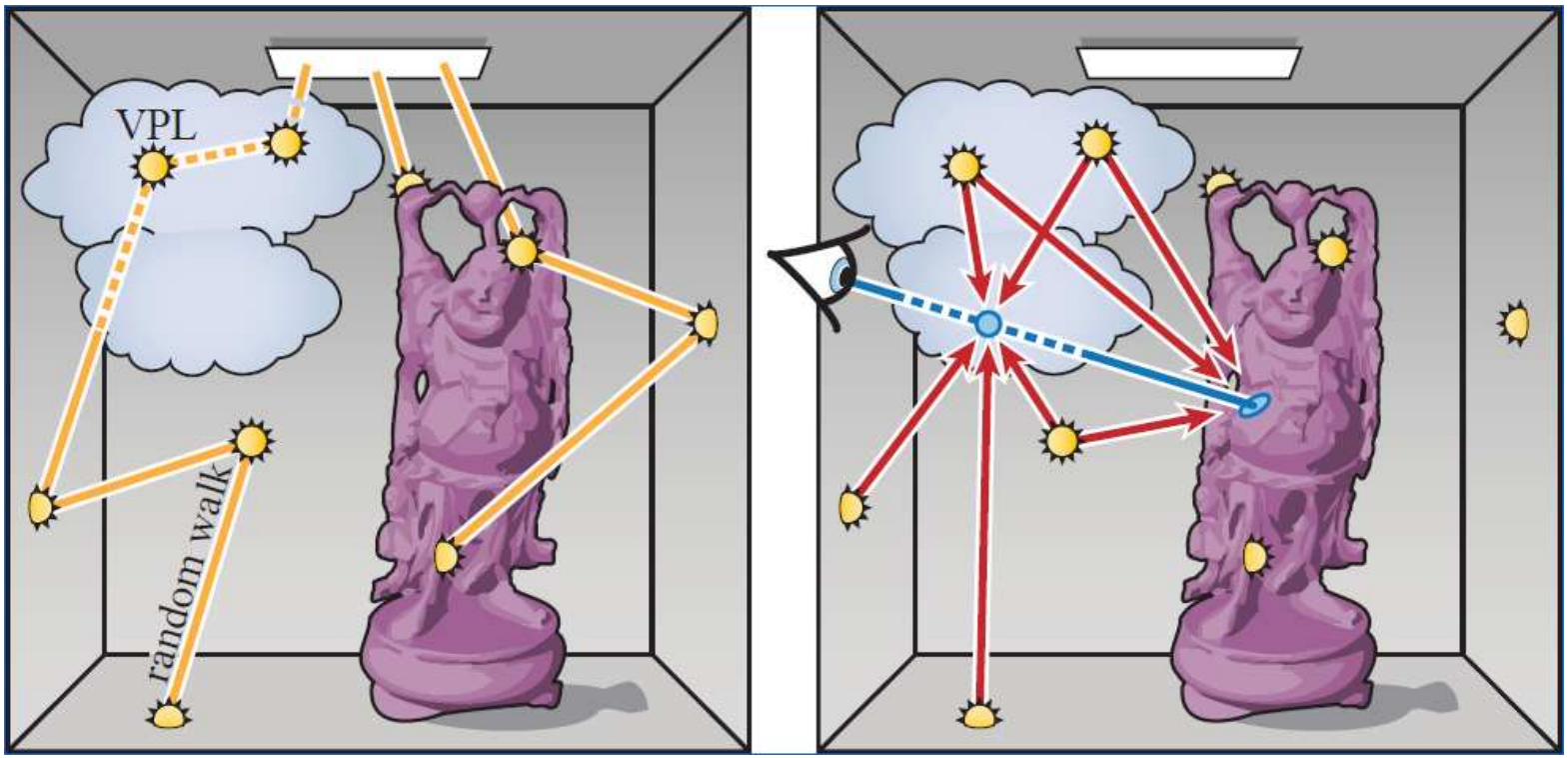



Directional occlusion
Directional occlusion + Bounce

Virtual Point Lights

two passes:

- spread some VPLs randomly bouncing through the scene
- calculate direct lighting from the VPLs



Virtual Point Lights

- full lighting transport simulation is reduced to the calculation of the direct illumination from many virtual light sources
- deferred rendering allows to shade from many light sources

Virtual Point Lights

- full lighting transport simulation is reduced to the calculation of the direct illumination from many virtual light sources
- deferred rendering allows to shade from many light sources
- not simple as it sounds
  - many details and problems to face
    - ★ path tracing, sampling, visibility computation...
  - many papers to study before I should even try to discuss them

1 Introduction

2 A modern OpenGL sandbox

3 Forward VS Deferred Rendering

4 Disk Based SSAO strikes back

5 Screen Space Global Illumination

6 Conclusions

Disk Based SSDO?

- what about bringing the disk based approach to SSDO?
  - sounds like the natural step forward
  - need to study SSDO details first, to avoid potential pitfalls
- probably some ideas can be taken from VPL based techniques too

TODO

- cleaning/refactoring the sandbox
- testing performances of my DSSAO implementation
  - compare results with the previous implementation
- studying more papers about the related subjects
- thinking about some improvement
- implementing it into the sandbox hoping it was a good one

TODO

- cleaning/refactoring the sandbox
- testing performances of my DSSAO implementation
  - compare results with the previous implementation
- studying more papers about the related subjects
- thinking about some improvement
- implementing it into the sandbox hoping it was a good one

"You can start your thesis!"