Interactive Cinematic Shading
Where are we?

Fabio Pellacini
Dartmouth College

Cinematic Rendering

- non interactive: [Pellacini et al 2005]
  - geometric complexity
    - 10^5 high-order primitives
    - 10^6 shaded points
  - shader complexity
    - ~10^3 individual shaders
    - ~10^5 instructions
    - ~5 GB textures

Cinematic Rendering

- artists want interactive preview

- previewing animation
  - crude models and appearance often good enough
- previewing lighting/shading
  - need highest quality possible
  - scalability in games has similar problems/solutions

Problem Statement

- given a production scene
  - geometry/shading specified in a prod. renderer
- render at *guaranteed* interactive rates
  - changing only shader parameters
  - fix view and geometry

Cinematic Relighting

- cinematic lighting is complex
  - major cost in movie production
  - poor interactive artists feedback

- relighting engines used for better feedback
  - take advance of static geometry and materials

Where are we?

- first systems appearing
  - tuned for particular productions
    - by efficiently supporting the right subset of possible shading
  - prove that high quality is possible interactively
  - used in real productions: [Pellacini et al. 2005]

Where are we? Far far away…

- designed for lighting operations only
  - but would like to extend to any shading
- require careful tuning of shaders
  - often manual simplification/translation
- do not support all lighting primitives
  - raytracing, global illumination, DSOs, …

Relighting Engines Primer

- fundamental assumptions
  - fixed geometry
  - fixed camera
- basic algorithm principle: deferred shading
  - precompute visibility from camera viewpoint
  - precompute non-changing partial shading values
  - recompute shaders as parameters are edited

Shading Slicing and Caching

- given a shader of the form

```
color shader(fixed[],changing[]) {
    caches[] = computeCaches(fixed);
    return computeResidual(caches, changing);
}
```

- compute and store caches
- for each change, re-execute residual shader

Example

- simple lighting model
  - Phong direct illumination
- cache material and geometry values
  - computed in surface shader
  - e.g.: position, normal, diffuse, specular, etc…
- compute each light using caching

Shader Slicing and Caching

- where to caches?
- what to cache?
- how to execute residual shaders?

Caching Domain

- problem: where to store caches?

- solution: object surfaces
  - mesh vertices or textures
- solution: image samples
  - image pixels
  - standard deferred shading

Image Space Caching

- pros: guaranteed framerate
  - does not depend on geometric complexity
- cons: aliasing
  - can only store one/few samples per pixels
    - otherwise caches become really large
  - hard to get handle hairs, motion blur, dof, etc…

Object Space Caching

- pros: allows for fully quality images
  - by using high quality filtering
  - exactly matches some renderers (Renderman)
- cons: recompute filtering
  - depends on geometry: cannot guarantee framerate
  - does not scale to fine geometry (hairs)
  - need a lot of samples for motion blur / dof

Caching Domain

- currently: image space caching
- research: faster object space and/or smaller image space

Shader Slicing

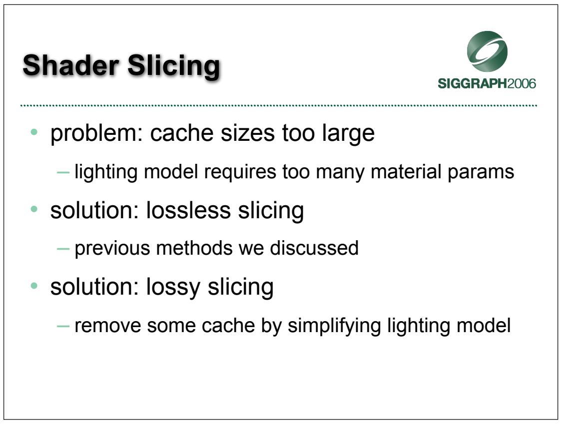
- problem: how to determine what to cache?
  - most shaders are not in the form shown before
- solution: manual slicing
  - artists write shader code in a “deferred form”
- solution: automatic slicing
  - compiler automatically determines what to cache

Manual Slicing

- pros: always works
  - code is written to match the caching mechanism
- cons: reduce coding flexibility
  - want to write code without worrying about caching
  - old shaders need to be rewritten
- cons: does not work if caching is changed
  - changing caching method might break the code

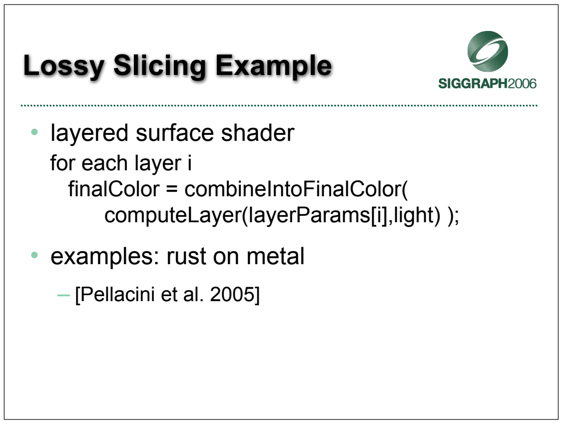
Automatic Slicing

- pros: compiler determines what to cache
  - [Guenter et al. 1995]
  - always correct
- cons: maybe not be optimal (memory)
  - store/compute ratio is hard to optimize
    - very important to reduce cache sizes



Shader Slicing

- problem: cache sizes too large
  - lighting model requires too many material params
- solution: lossless slicing
  - previous methods we discussed
- solution: lossy slicing
  - remove some cache by simplifying lighting model

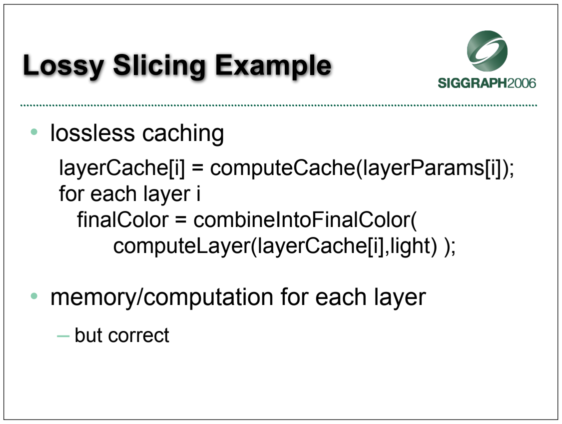


Lossy Slicing Example

- layered surface shader

```
for each layer i
   finalColor = combineIntoFinalColor(
         computeLayer(layerParams[i],light) );
```

- examples: rust on metal
  - [Pellacini et al. 2005]

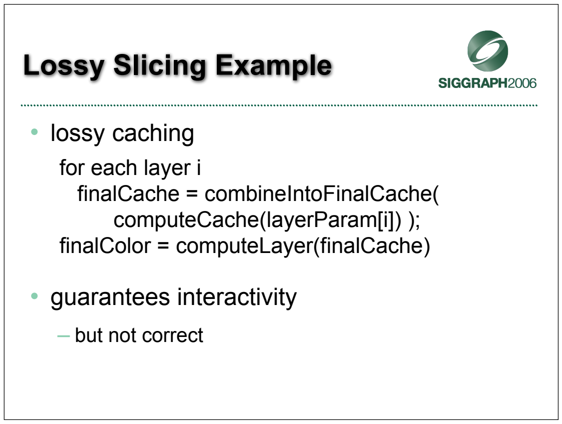


Lossy Slicing Example

- lossless caching

```
layerCache[i] = computeCache(layerParams[i]);
for each layer i
   finalColor = combineIntoFinalColor(
         computeLayer(layerCache[i],light) );
```

- memory/computation for each layer
  - but correct

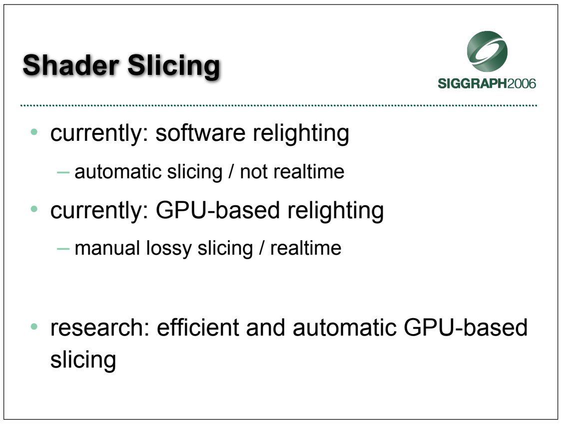


Lossy Slicing Example

- lossy caching

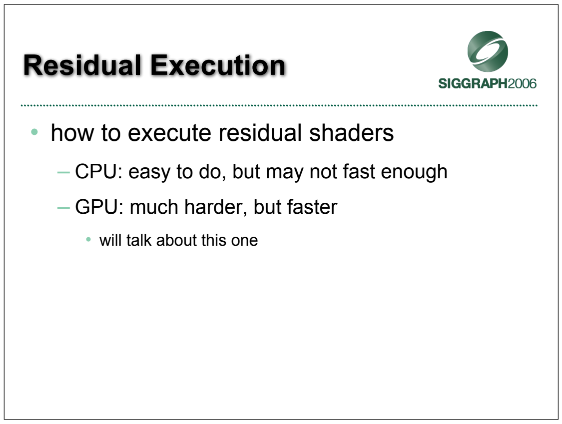
```
for each layer i
   finalCache = combineIntoFinalCache(
         computeCache(layerParam[i]) );
finalColor = computeLayer(finalCache)
```

- guarantees interactivity
  - but not correct



Shader Slicing

- currently: software relighting
  - automatic slicing / not realtime
- currently: GPU-based relighting
  - manual lossy slicing / realtime
- research: efficient and automatic GPU-based slicing



Residual Execution

- how to execute residual shaders
  - CPU: easy to do, but may not fast enough
  - GPU: much harder, but faster
    - will talk about this one

Residual Execution

- problem: residual shaders written in CPU language
  - how to translate them to GPU?
- solution: manual translation
  - artists manually create GPU version of shader
- solution: automatic translation
  - compiler translate shader versions

Manual Translation

- pros: works
- cons: takes time
- cons: does not scale to new GPUs
  - cannot adapt to new capabilities
  - same problem as games
  - but harder since lots of legacy shading code

Automatic Translation

- pros: compiler determines translation
  - various systems attempts to do this for Renderman
- cons: cannot support all CPU shading
  - does not know what to do in this case
  - covered later
- cons: might not be as efficient
  - computation structured differently on GPUs
  - CPU languages do not convey it well enough

Automatic Translation

- Renderman string operations
  - used for state binding: textures, matrices, etc.
- example: Ci = texture(“textureName”);

- problem: not a language transformation
  - GPU renderer has to load all possible textures

Automatic Translation

- Renderman shader plugins
  - any binary library that exposes interface
  - used heavily for all sort of things

- problem: cannot automatically translate
  - for example, allows for disk access from a shader

Automatic Translation

- Renderman derivatives
  - used to compute shading antialiasing

- problem: cannot automatically translate
  - unless GPU renderer has the same geometry than Renderman and uses multiple passes

Automatic Translation

- Renderman raytracing
  - used for shadow, reflection, indirect illumination
- problem: not supported efficiently on GPU
  - provide an external ray engine
    - e.g. [Pellacini et al. 2005]
- problem: requires synching with CPU while shading

Shader Translation

- currently: manual translation
  - automatic translation does not cover the language
- currently: language extensions for GPU
  - Renderman, MentalRay, …
- research: automate translation
  - more of an engineering/compiler problem though

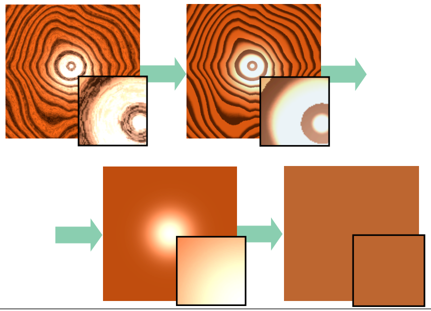
Residual Translation

- problem: residual might be too large
  - cannot guarantee interactivity
- solution: manual simplification
  - artists create shader simplifications
- solution: automatic simplification
  - automatically simplify shaders (not a compiler extension)

Manual Simplification

- pros: works somewhat
  - cannot tell how well it simplifies
- cons: (really) takes too much time
  - for large shaders, it is trial and error
- cons: (really) does not scale to new GPUs
  - performance evaluated on particular GPUs
  - same issue as game shader LODs

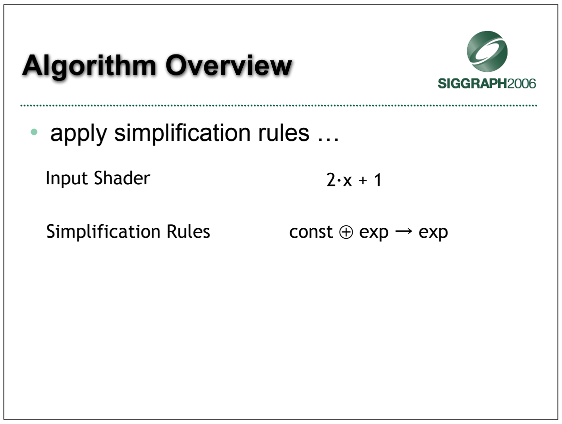
Automatic Simplification



Algorithm Overview

- input shader code

Input Shader 2·x + 1

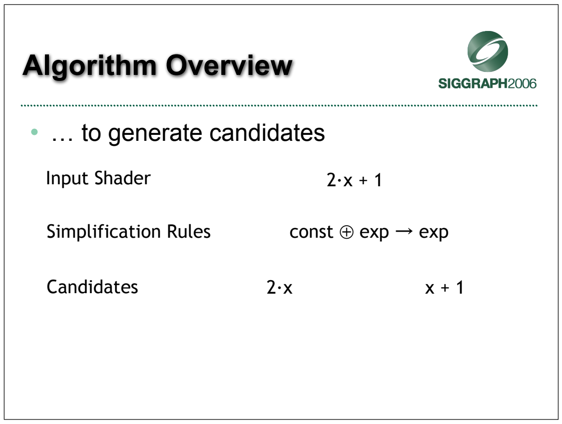


Algorithm Overview

SIGGRAPH2006

- apply simplification rules …

Input Shader	2·x + 1
Simplification Rules	const ⊕ exp → exp

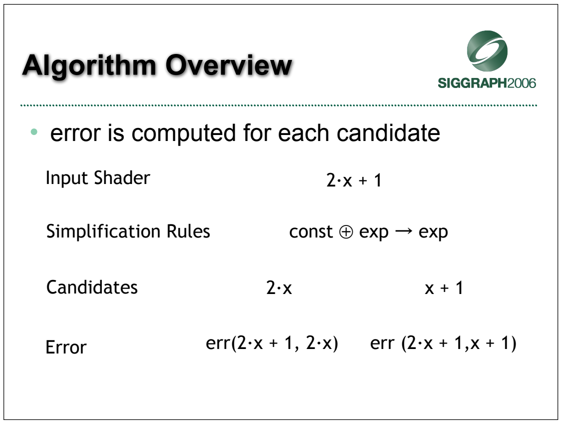


Algorithm Overview

SIGGRAPH2006

- … to generate candidates

Input Shader	2·x + 1	
Simplification Rules	const ⊕ exp → exp	
Candidates	2·x	x + 1

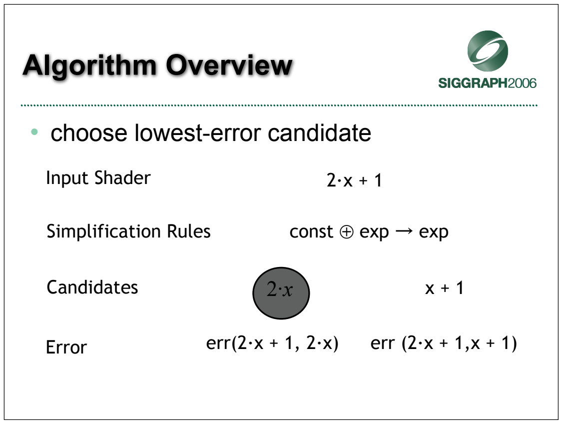


Algorithm Overview

SIGGRAPH2006

- error is computed for each candidate

Input Shader	2·x + 1	
Simplification Rules	const ⊕ exp → exp	
Candidates	2·x	x + 1
Error	err(2·x + 1, 2·x)	err (2·x + 1,x + 1)

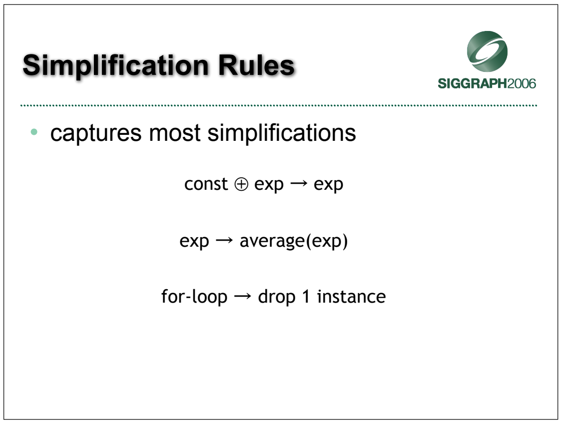


Algorithm Overview

SIGGRAPH2006

- choose lowest-error candidate

Input Shader	2·x + 1	
Simplification Rules	const ⊕ exp → exp	
Candidates	2·x	x + 1
Error	err(2·x + 1, 2·x)	err (2·x + 1,x + 1)



Simplification Rules

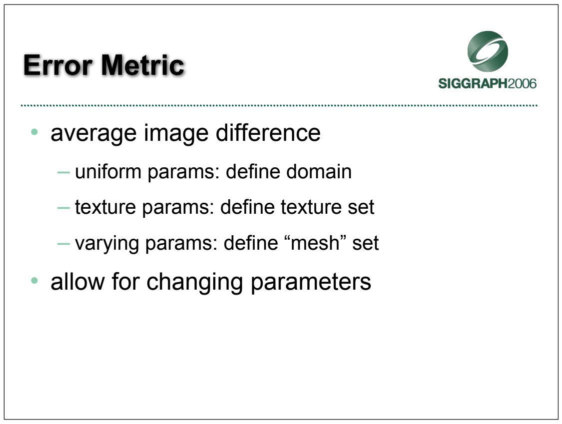
SIGGRAPH2006

- captures most simplifications

const ⊕ exp → exp

exp → average(exp)

for-loop → drop 1 instance

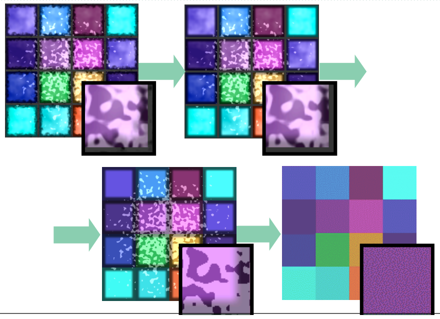


Error Metric

SIGGRAPH2006

- average image difference
  - uniform params: define domain
  - texture params: define texture set
  - varying params: define “mesh” set
- allow for changing parameters

Simplification Example



Automatic Simplification

- pros: determine “optimal” simplification
  - can try many more options than a person
- cons: does not scale to large shaders
  - not sure how close to the “best possible”
- cons: no solution to simplify data and code
  - [Olano et al. 2003] simplifies texture
  - [Pellacini 2005] simplified code
  - but hard to find a complete solution

Residual Simplification

- currently: manual simplification for GPUs
- currently: no simplification for CPU relighting

- research: better simplification
  - numerical-vs-structural

Where are we?

- realtime relighting is possible
  - manually translated/simplified shaders on GPUs
  - not many advanced lighting effects
    - but new work on the way
    - indirect illumination [Hasan et al. 2006]
  - essentially production “customized”
    - some approximations/solution only works for some productions

What can we not do?

- moving camera / dynamic scenes
- hairs / volumes
- really long and arbitrary CPU shaders
- dynamic indirect illumination

- this is future work for us research folks!

What did we learn?

- long/arbitrary shaders might not be needed
  - [Pellacini et al. 2005] shows that really simplified shaders look almost right

What did we learn?

- shaders do not express right abstractions
  - impossible to derive new algorithms since shaders are arbitrary
  - but are perfect for low-level GPU programming

What did we learn?

- (sadly) manual optimizations work
  - even for “simple” shaders: automatic translation, simplification, optimization not fruitful enough
  - they will never work for complex lighting/geometry effects (indirect, hair)
    - since it requires changing the algorithm, not the code

What did we learn?

- current goal: interactive renderer approximates offline

- better goal: offline renderer beautifies interactive
  - long term think about interactive rendering only
  - have the batch renderer make a “cleaner” picture