Software-Based High-Performance Graphics Computing

Philipp Slusallek
German Research Center for Artificial Intelligence (DFKI)
Intel Visual Computing Institute
Saarland University, Computer Graphics Group
Computer Science at the Saarland Campus

UNIVERSITÄT DES SAARLANDES

Multimodal Computing and Interaction

max planck institut informatik

Visual Computing Institute

Max Planck Institute for Software Systems
What is the Intel Visual Computing Institute?

- An open and collaborative research institute
  - Targeting basic research in Visual Computing
  - Special emphasis on many-core hardware
- A research institute of Saarland University
  - Funded by industry (Intel) and public agencies
  - Partners: Intel, UdS, DFKI, MPI-INF, MPI-SWS
  - Headed by two university professors
- Open for other members (industry and research)
  - European Hub for Visual Computing research
What will we be doing?

- Covering Entire Visual Computing Pipeline
  - Digital Signal and Media Processing
  - Markerless Geometry, Motion, Appearance Capture
  - Visual Simulation & Processing
  - Virtual Humans & Intelligent Behavior
  - Advanced Rendering and Visualization
  - Protocols for 3D Internet & Media Networking
  - Affective & Context-Based Visual Interaction
  - High-Performance Parallel Computing

Some of the hottest topics in the field
DFKI: Combining Graphics and AI

Agents and Simulated Reality
Philipp Slusallek

Intelligent Simulated Reality
Hilko Hoffmann

Living Lab: Saarland Visualization Center
Georg Demme

Computer Graphics Group
Saarland University

Safe and Secure Software
Werner Stephan

Multiagent Systems
K. Fischer/M. Klusch

Multimodal Computing and Interaction (TP7)
Cluster of Excellence
Multi Agents: Steel Mill „Saarstahl“
Multi Agents: Steel Mill „Saarstahl“

- Iron from blast furnace
- Preprocessing
- Steel converters
- Refinement
- Continuous Casting
- Further processing
Agents in Games: Intelligent Traffic Behavior

xaitment GmbH
Graphics Research Results
Realtime Ray Tracing

- Realtime Ray Tracing
  - Now in the main stream
- Fierce competition
  - Intel, Nvidia, ...
  - Building best HW for ray tracing
  - New many-core HW announced

The quality of 3-D computer graphics is poised for a quantum jump forward, thanks to speedier ways to simulate the flight of light

By W. Wayt Gibbs

For those of us who situated our formative years away blasting blocky space invaders, video games today can widen the eyes and slacken the jaw. The primitive pixelated ape of Donkey Kong has evolved into a three-dimensional King Kong of startling detail. Some new Xbox 360 games render their lead characters from an articulate mesh of more than 20,000 polygons, each tiny patch driven dozens of frames a second with its own subtle retrace, shading, and aliasing. Beyond the booming game industry, the evolution of graphics has lifted interactive software for design, engineering, architecture, medical imaging, and scientific visualization to new heights of performance. Much of the credit belongs to advances in graphics processing units (GPUs), the microchips at the heart of computer video cards that transform 3-D scenes into 2-D images at speeds faster than a trigger wince. As the rendering capabilities of GPUs soared, so did the revenues of ATI, NVIDIA and Intel, which make the most popular models.

A wide gulf of realism lurks, however, still separates interactive graphics from feature film effects and photography. And some experts say the only way that personal computers will ever cross that gulf—to reach the nirvana of computer graphics in which synthetic scenes display all the fluid motion and subtle shadings of reality—is through a basic change in how machines render 3-D models.

That change, from the so-called rasterization method that GPUs use to more scientific approach known as ray tracing, was long dismissed as infeasible for interactive, rapidly changing scenes. But advances in both software and hardware have recently propelled ray tracing to within range of the consumer PC market.

Ray tracing, which originated with Renaissance painter Albrecht Dürer and...
Intel Larrabee
Announcement @ IDF 2009
3D Internet

- Internet: THE communication network
  - Web browser: most used application, open all day
  - Everything converges to IP & Open Networks

- Everything?
  - There is still hardly any 3D content on the Web !!

- Does it matter?
  - 3D-HW: The other Mega-Trend of the last decade
  - Interactive 3D is the driving force for HW technology
  - All PCs/mobiles will contain many-core processors
  - Immersive 3D display for consumers (→ IFA)

- Ok, so what could we do with it?
Firefox with XML3D
Firefox with XML3D
Intelligent Simulated Reality

- **Reality**
  - Large-scale 3D models, highly detailed & realistic, ...
- **Simulated**
  - Illumination, acoustics, traffic, character animation, ...
- **Intelligent**
  - Semantics as a core feature
  - Multiagent systems, planning, speech, ...
- **Systems Oriented Research & Tools**
  - Build it from the ground up – reusing existing SW
- **Basis for the Future 3D Internet**
ISReal: System Overview

Realistic Realtime Rendering, 3D-GUI, Semantic Interaction, Tracking, and Immersive Visualization

Semantic World Model

P2P Middleware

Multiagent Simulation
Traffic Simulation
Human Simulation
Ontology Services
Animation Generator
Sensor Networks
Lighting Simulation
Acoustic Simulation
Hybrid Verification

Network

USER
Use Case: DFKI Smart Factory

Agent-Behavior:
- e.g. FSM
- plus
  - PDE,
  - Code to be executed,
  - ...

Verification View:
- CarriageAgent:
- velocity [in]
- stop/go [in]
- where [in, out]
- running [out]

X3D Object:
- Belt:
  - velocity [out]

X3D Agent:
- CarriageAgent:
  - velocity [in]
  - stop/go [in]
  - where [in, out]
  - running [out]

X3D Object:
- Carriage:
  - position [in]

PIM4Agent Metamodel

Hybrid System Verifier
- Elimination Approach
  - Integrated with
    - VSE
    - DocTIP

Integration approaches for model checking:
- PDE
- Code to be executed
- ...
New 3D Platform

- **Lightning**
  - Immersive user interface
- **RTSG**
  - XML3D based realtime scene graph
  - Also fastest X3D browser
- **RTfact**
  - Flexibility PLUS high-performance
  - Based on C++ meta-programming
- **New Programming Models**
  - High-performance many-core computing
Hardware has Changed !!!

Nvidia Fermi Architecture

Intel Larrabee Architecture

Massively parallel many-core processors

Future Processor Chips
Programming Issues

- HW architecture details are getting important
  - SIMD width, cache sizes & policies, mem latency, …
  - Have to change algorithms to match
- Inadequate programming languages
  - Too low level: E.g. memory layout is fixed in C++
  - Too high level: No memory control in Java
- Compilers know too little to generate good code
  - Automatic vectorization has large failed (for us)
  - No information about memory and communication
  - We do not know how to describe this, either !!
Options

- Keep doing the low-level work
  - No really an option
- **Domain specific languages**
  - Possible, but really, really hard – not recommended
  - Split-brain dilemma: working in multiple domains
  - Tend to become full languages anyway
- **Object-oriented design**
  - Has almost all the abstractions that we need
  - But need to improve efficiency and performance
- **Working on two approaches: RTfact and AnySL**
RTfact: Flexibility vs. Performance

- Experience with OpenRT ray tracing engine
  - Difficult to make changes (both algorithms or data)
  - Hard to modify for new technology
  - Inefficient for highly dynamic scenes
- Existing ray tracing systems
  - Trade-offs between flexibility and performance
    - Fast hand-coded low-level optimizations
    - Slow object-oriented designs
- Ideally: Need both flexibility AND performance
  - RTfact: How far we can push existing technology
RTfact: General Approach

- **Generic building blocks in C++**
  - Composable at design/compile time (templates)
  - Decoupling of algorithms and data structures
  - Multi-level abstractions using “C++ concepts”

- **Pragmatic and Compatible Approach**
  - Integrates with existing C++ tools and other SW
  - Known limitations: e.g. templates & virtual functions

- **High performance**
  - Code morphing through Template Meta Programming
  - Generation of fast code from large inlined blocks
RTfact: Generic Infrastructure

- Rendering
- Building
- Ray tracing
- SIMD Primitives
- Scene management
RTfact: Example Application

```cpp
PinholeCamera camera;  // initialization omitted
OpenGLFrameBuffer fb;  // initialization omitted

BasicScene<Triangle> scene;  // initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
                 DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
                    tree, intersector);
```
RTfact: Example Application

```cpp
PinholeCamera camera; // initialization omitted
OpenGLFrameBuffer fb; // initialization omitted
BasicScene<Triangle> scene; // initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
    DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
    tree, intersector);
```
RTfact: Example Application

```c++
PinholeCamera camera; // initialization omitted
OpenGLFrameBuffer fb; // initialization omitted

BasicScene<Triangle> scene; // initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
                  DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
                     tree, intersector);
```
RTfact: Example Application

```cpp
c_PinholeCamera camera;  // initialization omitted
OpenGLFrameBuffer fb;   // initialization omitted

BasicScene<Triangle> scene; // initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;

RayTracingRenderer<PixelCenterSampler,
                    DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
                    tree, intersector);
```
RTfact: Example Application

```cpp
PinholeCamera camera; // initialization omitted
OpenGLFrameBuffer fb; // initialization omitted

BasicScene<Triangle> scene; // initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
    DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
    tree, intersector);
```
RTfact: Example Application

```cpp
PinholeCamera camera; // initialization omitted
OpenGLFrameBuffer fb; // initialization omitted

BasicScene<Triangle> scene; // initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
    DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
    tree, intersector);
```
PinholeCamera camera;  // initialization omitted
OpenGLFrameBuffer fb;   // initialization omitted

BasicScene<Triangle> scene;  // initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
               DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
                     tree, intersector);
RTfact: Example Application

```cpp
PinholeCamera camera;        // initialization omitted
OpenGLFrameBuffer fb;        // initialization omitted

BasicScene<Point> scene;     // initialization omitted
LoDKdTree<Point> tree;
LoDKdTreeBuilder builder;
LoDKdTreeIntersector<PointIntersector> intersector;
RayTracingRenderer<PointCenterSampler,
                      LoDIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());
renderer.render<16>(scene, camera, fb, fb.getClipRegion(),
                     tree, intersector);
```
## Result

<table>
<thead>
<tr>
<th>Method</th>
<th>Sponza</th>
<th>Conference</th>
<th>Soda Hall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet kd-tree</strong></td>
<td>4.5</td>
<td>4.2</td>
<td>5.1</td>
</tr>
<tr>
<td>OpenRT</td>
<td>4.7</td>
<td>4.2</td>
<td>5.4</td>
</tr>
<tr>
<td>Manta</td>
<td>4.7</td>
<td>4.2</td>
<td>5.4</td>
</tr>
<tr>
<td>RTfact</td>
<td>6.8</td>
<td>6.4</td>
<td>6.5</td>
</tr>
<tr>
<td><strong>Frustum BVH</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DynBVH</td>
<td>N/A</td>
<td>9.3</td>
<td>11.1</td>
</tr>
<tr>
<td>Manta</td>
<td>4.5</td>
<td>4.8</td>
<td>5.6</td>
</tr>
<tr>
<td>Arauna</td>
<td>13.2</td>
<td>11.3</td>
<td>N/A</td>
</tr>
<tr>
<td>RTfact</td>
<td>13.1</td>
<td>11.6</td>
<td>11.4</td>
</tr>
</tbody>
</table>
RTfact: Conclusions

- **Great success**
  - Most flexible and really fast ray tracing engine
  - Solid basis for much of our current work
    - BMBF Avilus, EU Vision, BMBF ISReal, BioInfo, IVCI, FhG 3D Reconstruction, ...

- **Limitations**
  - Compile-time composition
  - Limited expressiveness of templates and C++
  - Practical issues: Error message, compile time, ...

- **Need to have more control over the compiler**
  - Higher level abstraction but control mapping to HW
**AnySL: Shaders**

- **Programmable Shading**
  - Allow to change core rendering features
  - RenderMan by Pixar [Hanrahan 1990]
    - The mother of all shading languages
  - Rasterization:
    - Specific to pipeline model (split into vertex and fragment)
    - Cg (Nvidia only), HLSL (DX only), glsl (OpenGL only)

- **Shader: A plugin for the innermost loops**
  - Run for every new ray, surface hit, light sample, ...
  - For volume rendering: For every MADD along ray
  - For one-liners to thousands of LOCs
Example Shader: Wood

- **Using RenderMan SL**

```cpp
surface
wood(float ringscale = 10;
    color lightwood = color(0.3, 0.12, 0.03),
        Darkwood = color(0.05, 0.01, 0.005);
    float Ka = 0.2, Kd = 0.4, Ks = 0.6, roughness = 0.1)
{
    point NN, V, PP;
    float y, z, r;

    NN = faceforward(normalize(N), I);
    V = -normalize(I);
    PP = transform("shader", P);
    PP += noise(PP);
    y = ycomp(PP);
    z = zcomp(PP);
    r = sqrt(y * y + z * z) * ringscale;
    r += abs(noise(r));
    r -= floor(r);
    r = smoothstep(0, 0.8, r)
        - smoothstep(0.83, 1.0, r);
    Ci = mix(lightwood, darkwood, r);
    Oi = Os;
    Ci = Oi * Ci * (Ka * ambient() + Kd * diffuse(NN))
        + (0.3 * r + 0.7) * Ks * specular(NN, V, roughness);
}
```
Why Are Shaders Good?

- **Portability**
  - Hide implementation details of the rendering system from the programmer

- **Convenience**
  - Provide graphics-related language constructs (illuminance loop, vector/matrix data types)

- **Restrictiveness**
  - Prevent the programmer from using obscure language features

- **Performance**
  - Code can be customized to rendering system
From Theory to Practice ...

- **Portability**
  - Not really: often tied to a certain rendering model

- **Convenience**
  - Special constructs can be emulated
    - Classes, operator overloading, templates, ...

- **Restrictiveness**
  - Newer SLs are more and more permissive. Why reinvent C++?

- **Performance**
  - Mismatch between shader and renderer environment
  - Need to be more tightly integrated
From Theory to Practice …

- Existing compilers targeted at one specific renderer
  - Rasterization (Cg, HLSL, glsl)
  - RenderMan (Reyes)
- If you build a new renderer and want shader support
  . . . you end up building a compiler
Any SL: From ABIs to an Embedded Compiler

- Traditional Approach
  - Fixed ABI for dynamic loading of code/data
  - Many issues
    - Many virtual function calls, indirection, fixed interface, …
    - No common optimization (in inner kernel !!)
    - No ability to transform code (e.g. packetization)
AnySL: From ABIs to an Embedded Compiler

Renderer

Data
Code

Glue Code

API / ABI

Shader DSO/DLL

API / ABI Spec
Shader Data
Shader Code

Compiler (LLVM)

Renderer

Data
Code

Glue Code

API
Shader Data
Shader Code

Compiled Shader

Compiler (LLVM)

Data
Code
Optimized Shader

Compiler (LLVM)
Example: Packet-Based Ray Tracing

- Modern ray tracers shoot packets of rays

  - Exploit SIMD instructions of modern CPUs
    - Can execute instruction on \( k \leq n \) floats at once
    - Current architectures:
      - SSE: \( k = 4 \)
      - AVX: \( k = 8 \)
      - Larrabee: \( k = 16 \)
  
- Shader has to shade \( n \) hit points at once
AnySL: Packetized Shaders

- Writing packetized shaders is really HARD
  - Not an option for any application

- Traditional:
  - A shader is given by a control-flow graph of scalar instructions

- Needed:
  - A packetized shader is a new shader that executes $k$ instances of the original shader at once

- Control flow of instances can diverge!
Example Control Flow

- Diverging control flow of a shader
  - Need to efficiently merge flows again!

- Shaders are nested in a deep recursion
  - Must handle closures and reordering of packets
Packetized Shaders

- **Approach**
  - Program transformation
  - Flatten control flow
  - Every instance executes all instructions
  - Mask out wrong results
  - Loops are iterated until last instance is done
  - Already exited instances are invalidated
  - Simulate what GPUs do in HW

![Diagram](image_url)

<table>
<thead>
<tr>
<th>Inst</th>
<th>Executed blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>2</td>
<td>1 2 3 4 5 2 3 4 5 6</td>
</tr>
<tr>
<td>3</td>
<td>1 2 3 4 5 2 3 4 5 6</td>
</tr>
<tr>
<td>4</td>
<td>1 2 3 4 5 2 3 4 5 6</td>
</tr>
</tbody>
</table>

Load: 26
Utilization: $26/40 = 65\%$
Speedup: $26/10 = 2.6x$
Packetized Shaders

• You may not want to do this by hand

```
if (all(inside(voc2(u.x, v.x), 0.f, 1.f))) {
    vec3 p0 = mix(mix(p00, p01, v.z),
                   mix(p10, p11, v.x), u.x);
    ...
}
```

```
__m128 u_x = _mm_div_ps(_mm_xor_ps(_mm_add_ps(_mm_mul_ps(Av1, v.x), A1),
                          _mm_casti128_ps(_mm_set1_epi32(0x80000000)));
__m128 __m128 u_y = _mm_div_ps(_mm_xor_ps(_mm_add_ps(_mm_mul_ps(Av1, v.y), A1),
                          _mm_casti128_ps(_mm_set1_epi32(0x80000000)));

...`
```

• Compilers
  - Cannot optimize well on flattened control flow
  - Optimize scalar version, then packetize
Packetized Shader

- Results of Master thesis (R. Karrenberg)
  - Packet size $k = 4$
  - Completely automated (LLVM)
  - Shaders written “scalarly”
  - “Packetized” automatically
  - On average 3.2x speedup
  - Sometimes superlinear $\rightarrow$ locality
  - Not specific to graphics
  - Can be used wherever data parallelism is available

<table>
<thead>
<tr>
<th>Scene</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>brick</td>
<td>4.0x</td>
</tr>
<tr>
<td>checker</td>
<td>4.5x</td>
</tr>
<tr>
<td>checker2</td>
<td>4.8x</td>
</tr>
<tr>
<td>glass</td>
<td>5.0x</td>
</tr>
<tr>
<td>glass2</td>
<td>5.7x</td>
</tr>
<tr>
<td>granite</td>
<td>1.0x</td>
</tr>
<tr>
<td>parquet</td>
<td>1.3x</td>
</tr>
<tr>
<td>phong</td>
<td>2.7x</td>
</tr>
<tr>
<td>screen</td>
<td>4.9x</td>
</tr>
<tr>
<td>starball</td>
<td>2.5x</td>
</tr>
<tr>
<td>starball2</td>
<td>3.0x</td>
</tr>
<tr>
<td>whitted</td>
<td>4.5x</td>
</tr>
<tr>
<td>whitted2</td>
<td>4.8x</td>
</tr>
<tr>
<td>wood</td>
<td>2.0x</td>
</tr>
</tbody>
</table>
Results
AnySL: Conclusions

Summary

- Shaders are compiled to platform-independent machine code
- Can be produced from any shading language
- Or other languages like C++
- Highly-optimizing JIT compiler within the renderer
- Reduce work for the renderer implementor:
  - Provides API adapters that are optimized away
  - Platform-independent packetization
- Significant speedup on benchmarks (3.2x)
AnySL: Future Work

- Support other hardware targets & platforms:
  - HW: GPUs, Larrabee; SW: other front/back ends
- Shader debugging & visualization
- Advanced, high-quality features:
  - Derivatives: Automatic differentiation of code
  - Bounds: Generate code for affine arithmetic
- Ray tracing as stream programming?
  - Express the renderer as net of shaders
  - Exploit more parallelism ⇒ create more coherence
- Adaptive multi-level compilation
Outlook: Beyond Graphics

- A new model for efficient plugins
  - May enable much more interesting apps
    - E.g.: Program your own filters in Photoshop?

- Provide Programmer Tool for Transformations
  - Library of program transformations
  - Democratization of Compilers
    - The programmer knows best

- Avoiding Domain Specific Languages
  - They use new syntax to provide new semantics
    - But: Syntactic expressiveness may be good enough
  - App provides compiler with additional semantics