Creating Coherence: Ray Tracing, Spatial Search, and Irregular Data Structures

Michael McCool

August 9-10, 2008
IRT, Los Angeles

RAPIDMIND

Agenda

- Background
  - RapidMind platform
  - Portability and many-core processor commonalities
- Ray Tracing
  - A driving application for platform design
  - Problem structure and challenges
- Generalization
  - Related applications in spatial search
  - Solution strategies
- Coherency
  - Data structures
  - Control flow

Overview

- ***Portable software development platform for multi-core and many-core processors***
- Single-source solution for portable high-performance parallel programming
- Supports high productivity development
- Safe and deterministic general-purpose programming model (SPMD stream)
- Scalable to an arbitrary number of cores
- Can be used to target **both** accelerators and multi-core processors
- Integrates with **existing** C++ compilers

Explicit Parallelism and Locality

Usage:

- Use existing ISO C++ compiler
- Include header, link

Data:

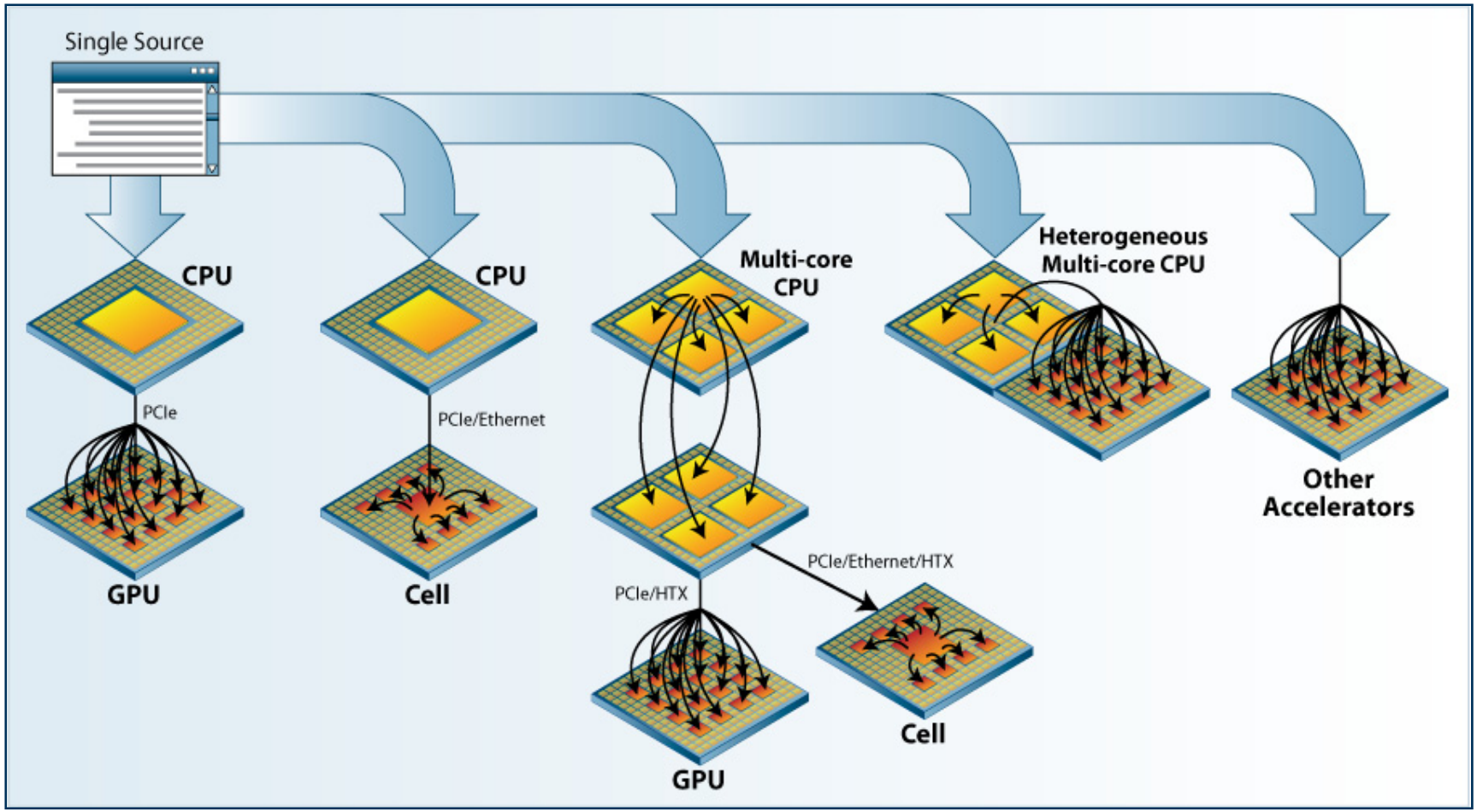
- Tuples
- Arrays
- *Global data abstraction*

Programs:

- Defined dynamically
- Execute on coprocessors
- *Remote procedure abstraction*

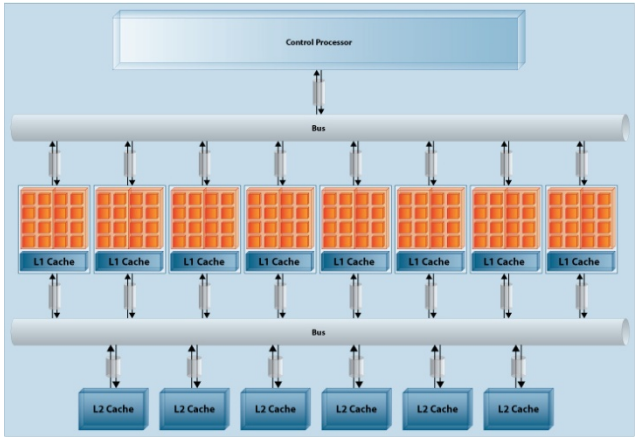
```
#include <rapidmind/platform.hpp>
using namespace rapidmind;

Value1f f = 2.0f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

Program prog = BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = (r + s) * f;
} END;

a = prog(a,b);
f = 3.0f;
stride(a,2,2) = prog(
   slice(a,0,255,0,255),
   slice(b,256,511,0,255));
```

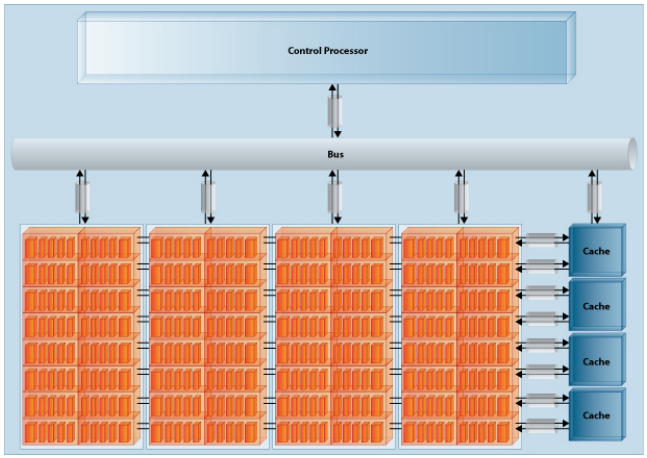
Portability
Present and Future Benefits

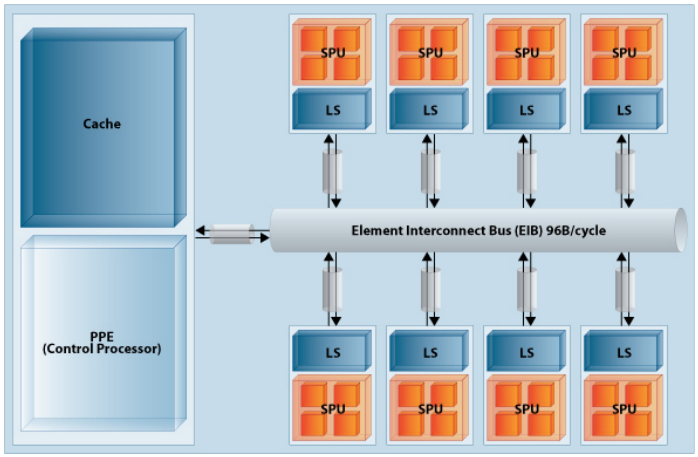


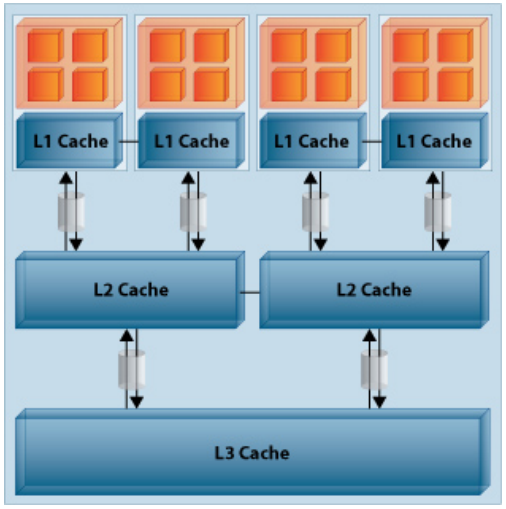
Provides deployment choice now, but also future proof

Architectures: GPU, Cell, CPU









Obtaining Performance

Compute

- Exploit parallelism
- Run on many-core vector processors including GPUs, Cell, CPUs
- Both multi-core and vector parallelism mechanisms
- Needs to be efficient and scalable

Data

- Hide latency (use even more parallelism…)
- Exploit coherence
  - Vector and cache alignment
  - Block transfers
  - Temporal and spatial locality
- ***If coherence doesn't exist, it is necessary to create it***
  - ***How?***

A Driving Application
Ray Tracing

Real-time ray tracing
Reflection and refraction
Many recursive rays per pixel
Full shader and texture support

Commercial product:
Developed by RTT AG, Germany
Automotive CAD visualization

Real-time since 2005

Problem Structure

1. Creating and updating accelerator structure
2. Computing nearest ray/primitive intersections

Essentially a "spatial search" database problem:

1. Create and update index
2. Perform queries

With a couple of complications:

➔ Queries can generate secondary queries

➔ Also need to access textures from shaders

Related Applications

Many applications of spatial databases:

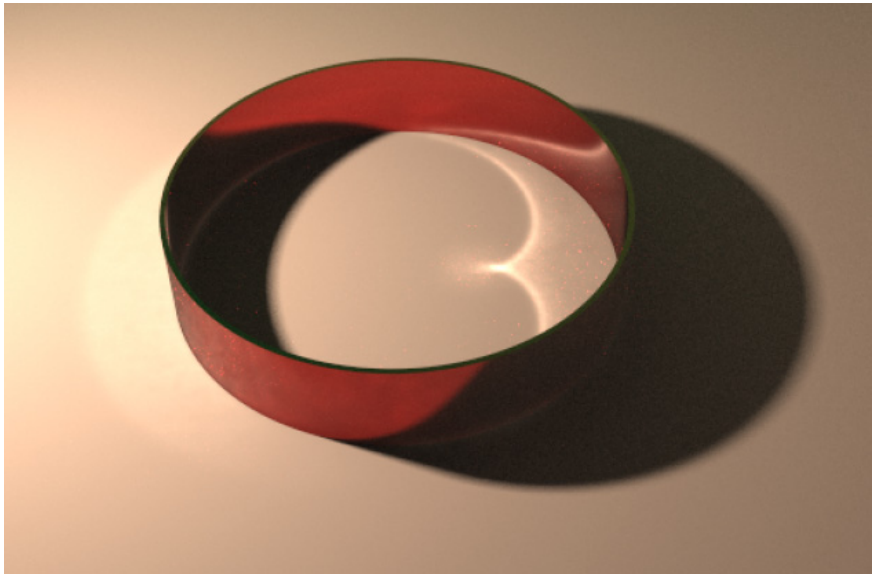
- Flocking
- Photon mapping
- Collision detection
- Textures
  - Sparse textures
  - Lossless variable-rate compression with random access
  - Vector graphics textures
- Search
  - Relational and text
  - Sequences, signals, and images
- Fluid flow
  - Particle dynamics

Flocking

- “Boids” each have state including position, direction, velocity, moods…
- Each “perceives” state of k nearest neighbours
- Executes rule to update its own state
- Attraction/repulsion → emergent collective behaviour

Photon Mapping

- Scatter photons into the scene by path tracing from light sources
- Render from eye
- To shade surface points, estimate irradiance using density estimation based on k nearest neighbours at each query point



K Nearest Neighbour Problem

Database: set of points in n-D space

Query: set of points in n-D space

For every point in query set, find k points closest to each in the database.

Observations:

- For graphics, often n equals 2 or 3
- Approximate solution is useful in *some* applications
- *Some* data has a minimum separation distance
- In *many* cases, query and database are identical

K Nearest Neighbour Algorithms

1. Grid search
   - Spatial subdivision
2. Kd-trees
3. Locality sensitive hashing
4. Hilbert R-trees
   - Bounding volume hierarchy
   - Gives very fast build, but non-optimal from SAH point of view

Grid Search

Index structure: grid of lists

Algorithm:

1. Look up cell holding query point
2. Search it and its neighbours until k points found

Observations:

- Uniform grid inefficient if points unequally distributed
  - Too close together: long lists in each grid, serial search
  - Too far apart: search large number of neighbours
- Parallel update of data structure troublesome
  - Read-modify-write to insert element into grid cell
- Multiresolution grid can address some of these issues

Kd-Trees

Index structure: Kd-trees

Algorithm:

1. Build Kd-trees using suitable splitting heuristic
2. For each query point, descend to nearest leaf and initialize bounding radius
3. Backtrack recursively to eliminate all points outside that radius (and include closer ones)

Observations:

- Building optimal Kd-tree difficult
- Tree generally not balanced, so requires pointers
- Backtracking inefficient without suitable caching, local store

Locality Sensitive Hashing

Index structure: hash table(s)

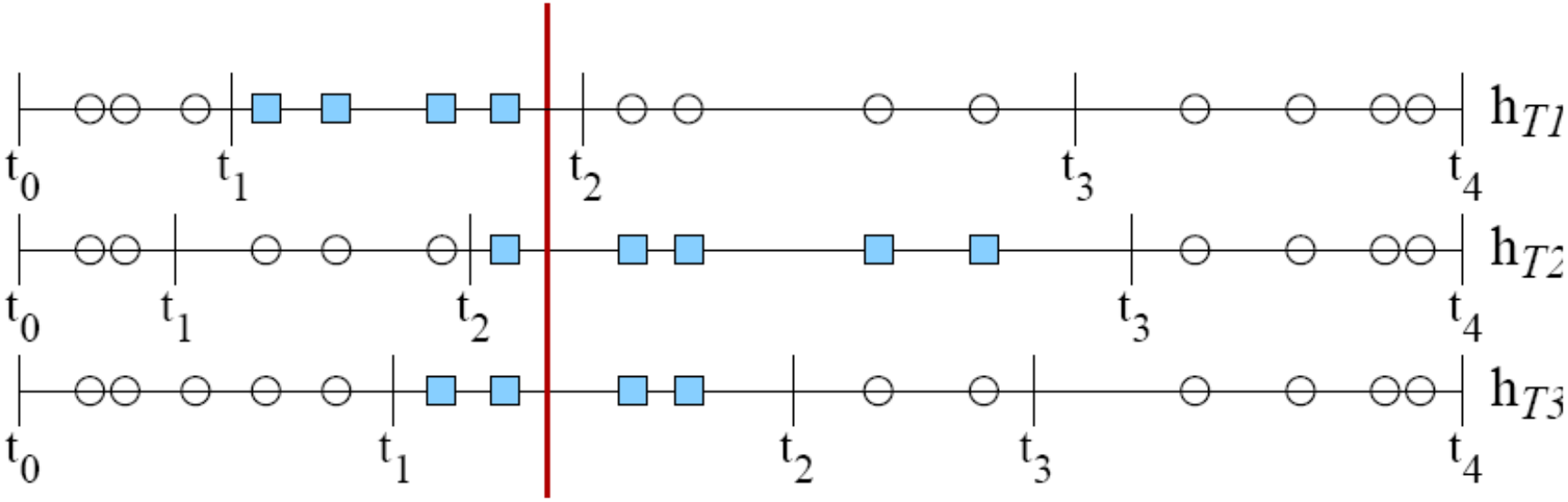
Algorithm:

1. Insert all database points into set of hash tables
2. Map query through set of hash functions
3. Search over nearby points in each hash table
4. Remove duplicates

Observations:

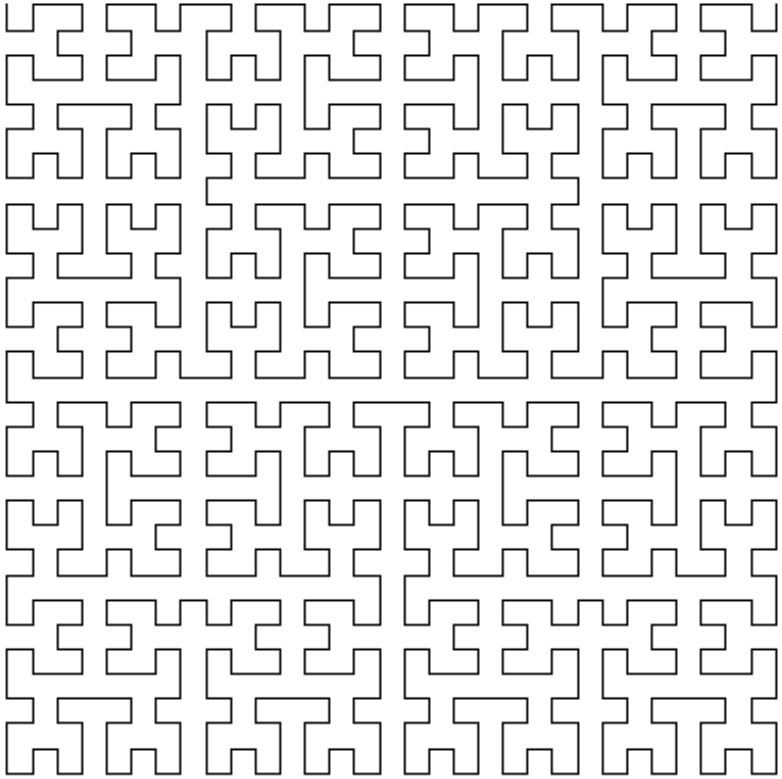
- Most suitable for high dimensional problems
- *Approximate.* This is ok in some applications, but not in many. In particular, it causes artifacts in photon mapping.

Locality Sensitive Hashing

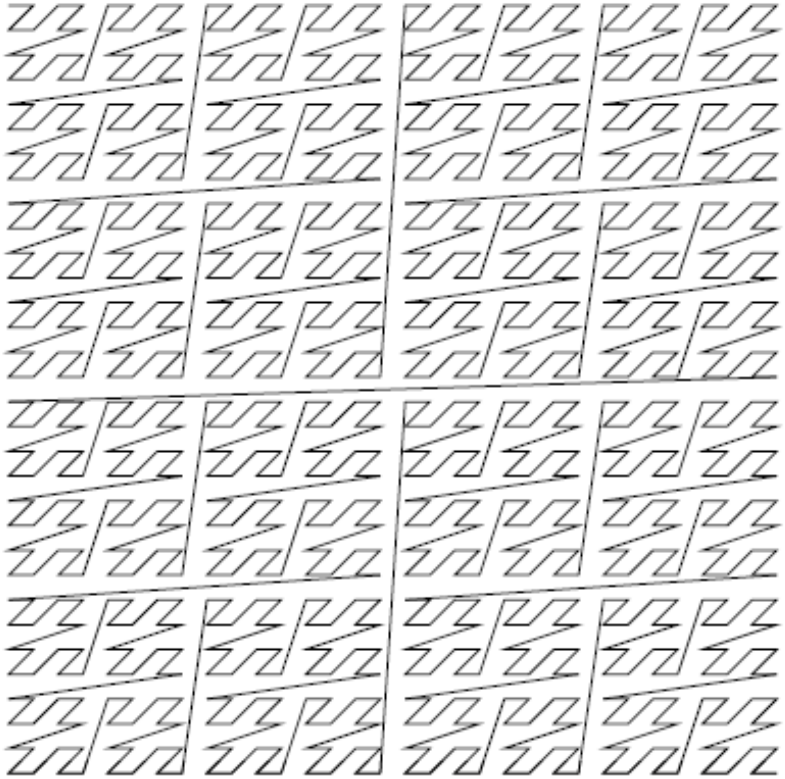


Space Filling Curves

- Invertible transformations between 1D and nD space



(a) Hilbert Curve



(b) Z−order Curve

Hilbert R-Trees

Index structure: balanced R-tree (BVH)

Algorithm:

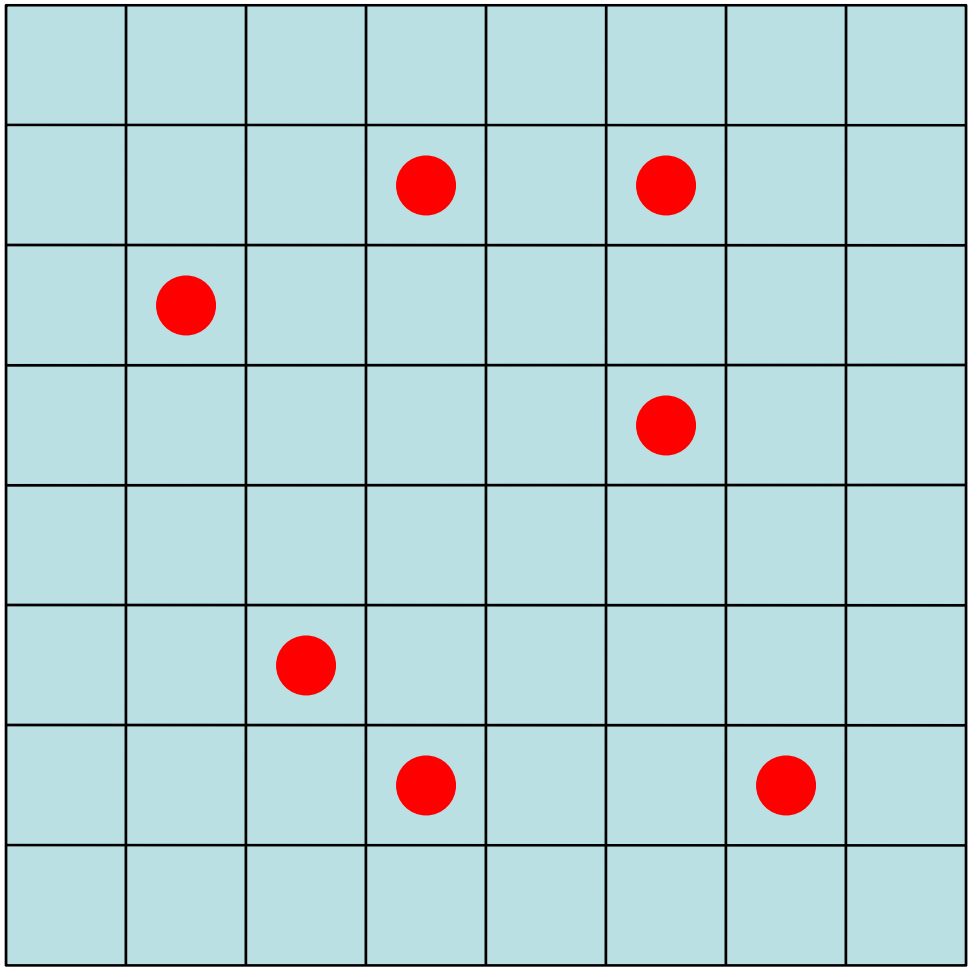
1. Map query points into Hilbert order, sort, split
2. For each query point, descend to nearest leaf and initialize bounding radius
3. Backtrack recursively to eliminate all points outside that radius (and include closer ones)

Observations:

- No pointers needed; balanced tree; but NOT SAH optimal...
- Hilbert map form of locality preserving hash
- Backtracking inefficient without suitable cache, local store

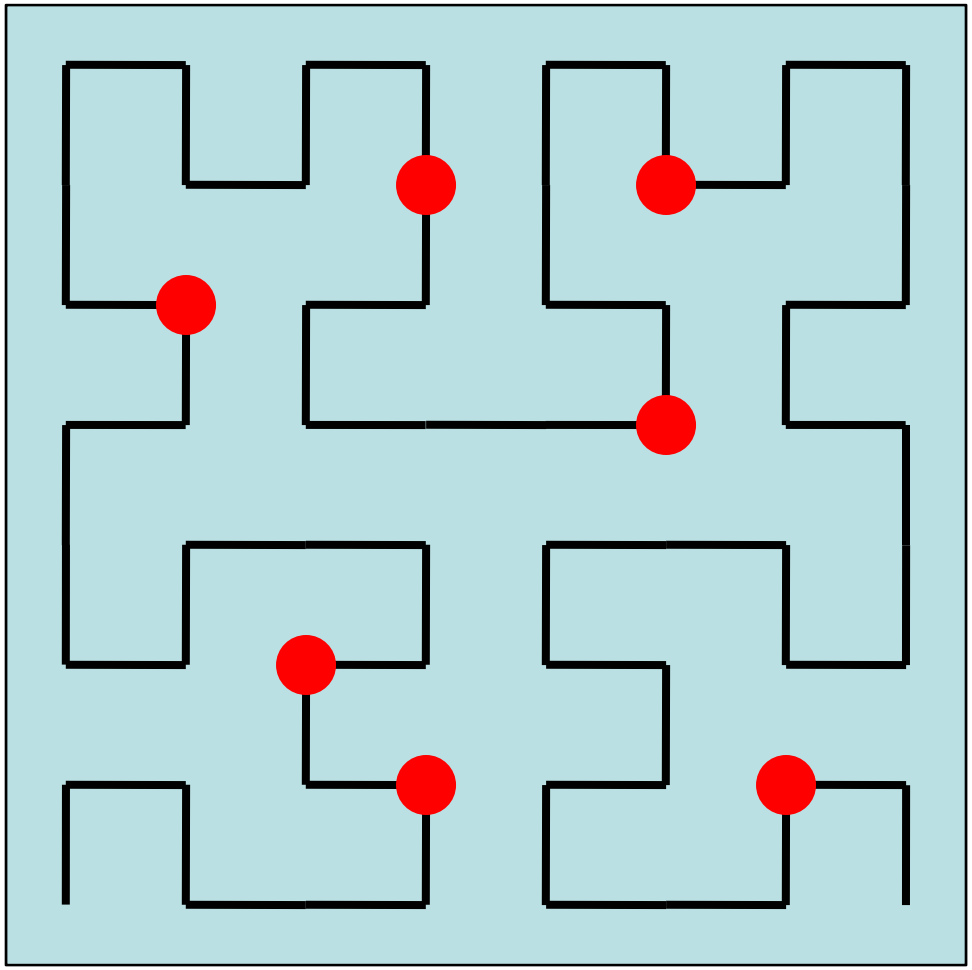
Hilbert R-Trees

- Given a set of points in nD space
- Quantize positions to integer power-of-two grid



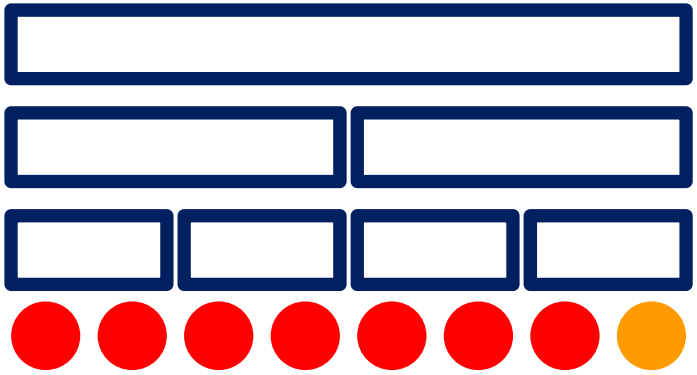
Hilbert R-Trees

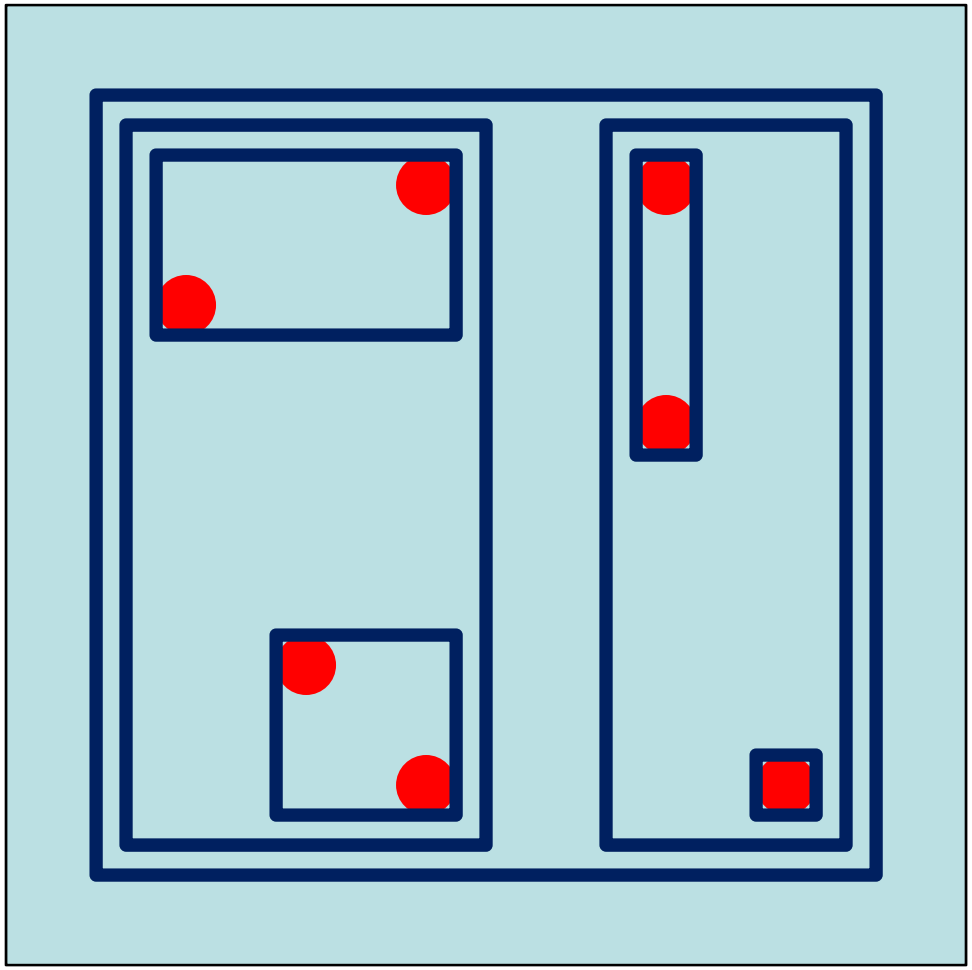
- Compute Hilbert index of each point
- This is distance of each point along curve
- Can be computed with simple state machine



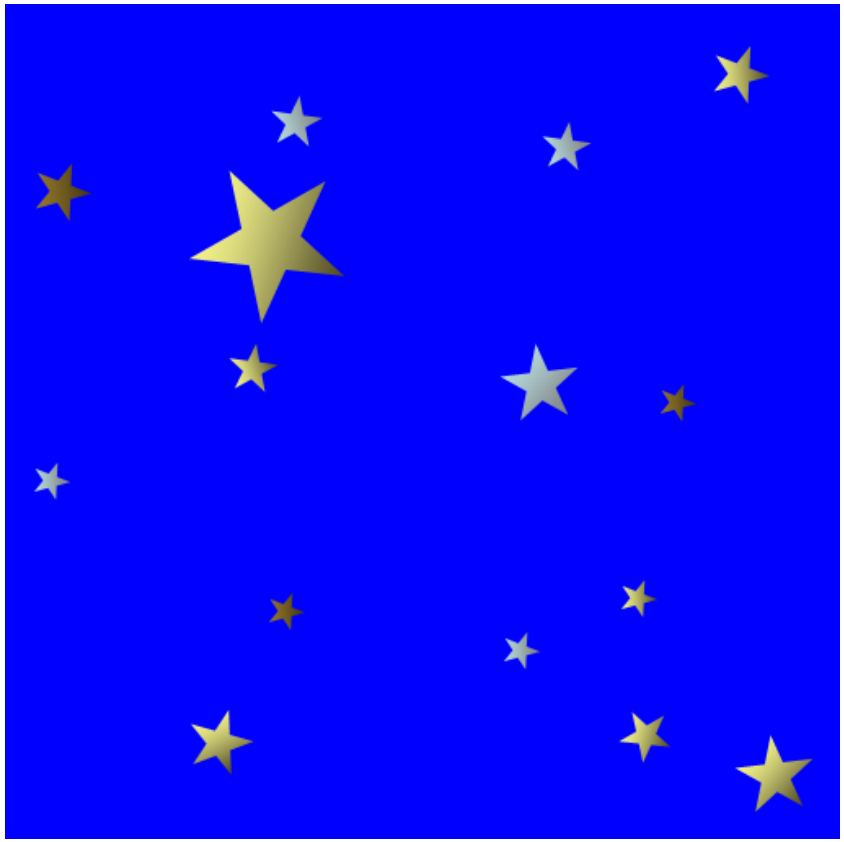
Hilbert R-Trees

- Sort by index, then recursively bound
- Gives balanced R-tree (but non-optimal)
- Can be stored as sequence of arrays





Sparse Textures



Problem

Database: Sparsely populated array

Query: given point in space, look up data for that point or, if empty, use “background” value

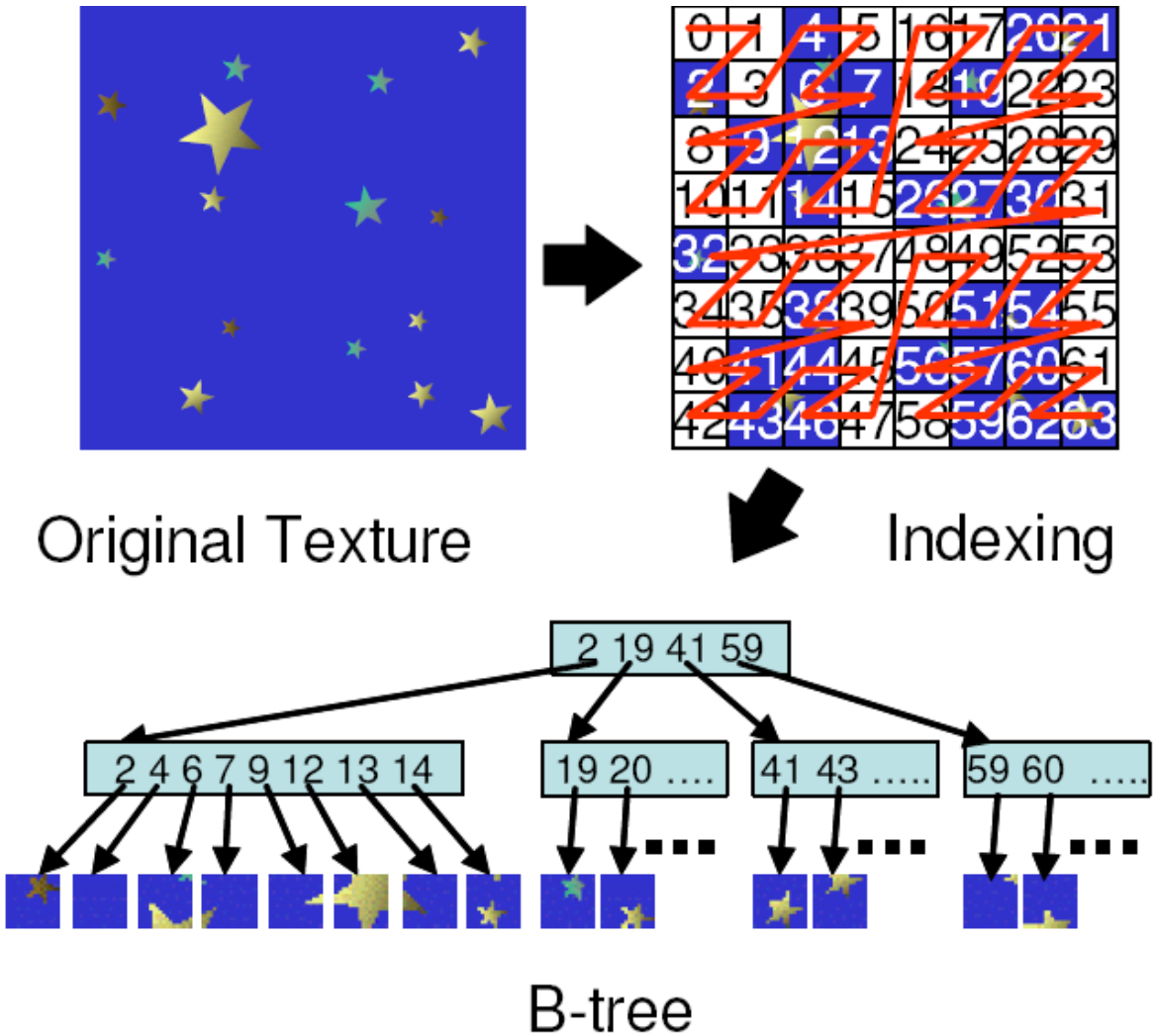
Observations:

- Conceptually consistent with regular textures
- Efficient sparse textures would eliminate need for texture atlas packing
- Efficiently stored (and updateable) sparse arrays could be substituted for many other data structures
  - *Equivalent to associative arrays*

Representations

1. Grid
   - Two-level index
   - One level of indirection into packed array of tiles
   - Can only handle sparsity
2. Multiresolution grid
   - Two-level index
   - As above, but with scale factor in index and variable-sized tiles
   - Variable sized tiles need to be packed
3. B-Tree
   - Multi-level index based on space-filling curve mapping
   - Packing leaves supports both uniform-sized block transfers and variable-size tiles
   - Supports efficient insertion

B-Tree Representation

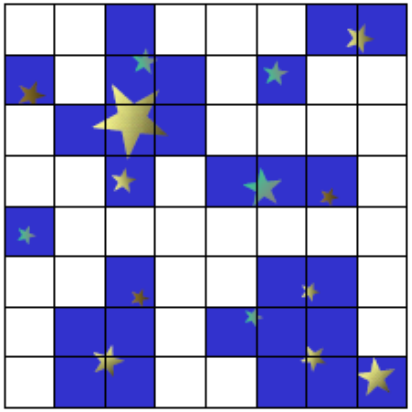


B-Tree

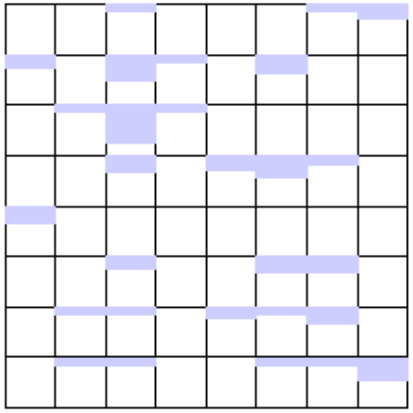
- Block-oriented tree structure
  - Good cache and memory access behaviour
- Every node has between n/2 and n children
  - Sorted key/pointer pairs
- Insertion:
  - Create new leaf node
  - Locate node at bottom of tree for insertion
  - If node not full, insert in sorted order
  - If node full, split, and insert pointer to new index node in parent
- Always results in balanced tree
- Deletion (while maintaining balanced tree) harder…
  - But in many applications can just throw away entire tree

Variable-Rate Lossless Compressed Textures
Representation and Compression

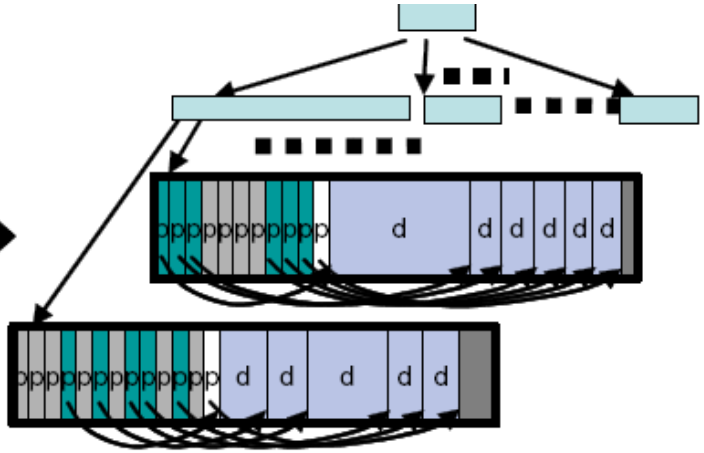
Original Texture



Occupied Tiles



Difference-based Compression

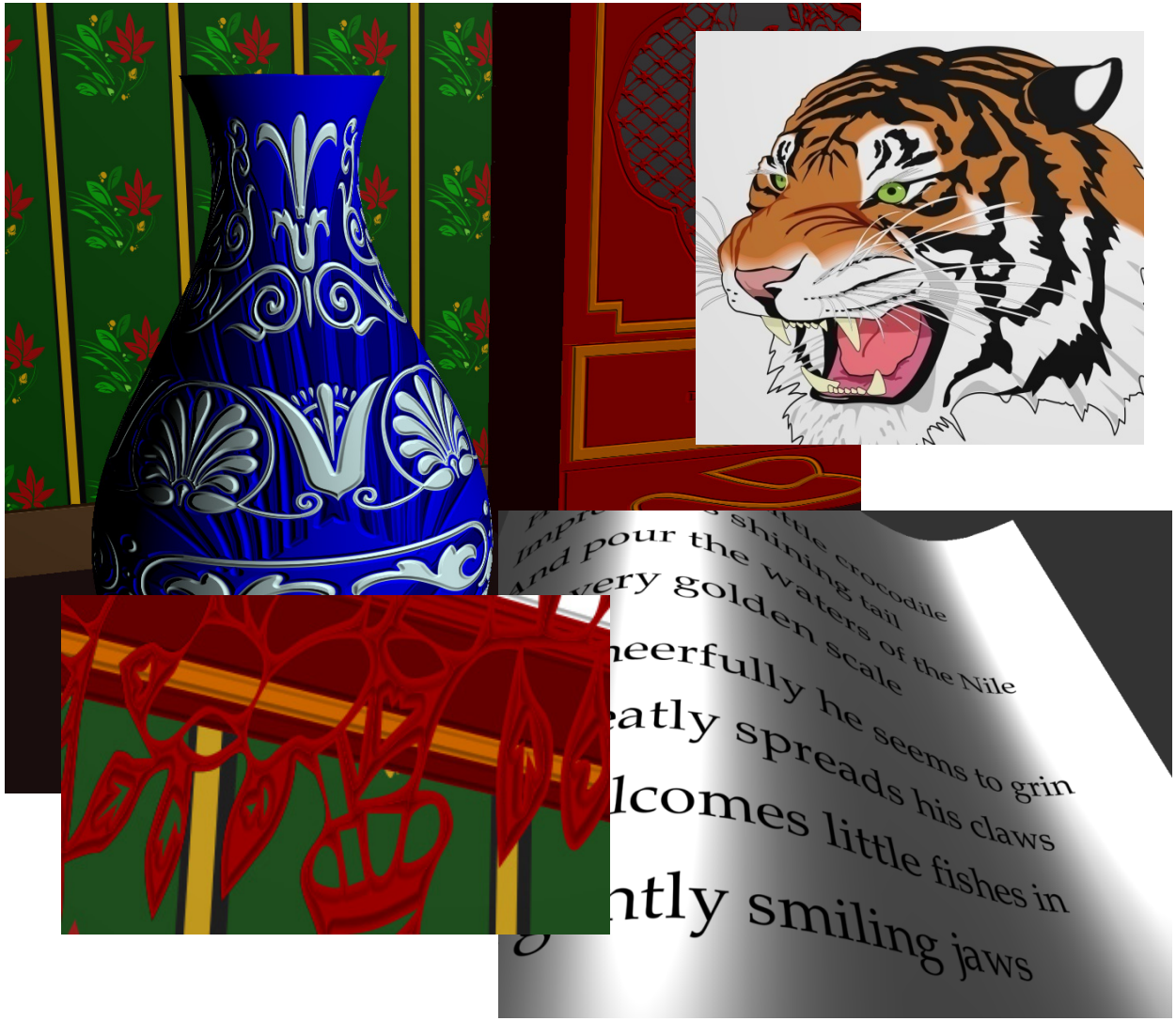


Index Structure

Vector Textures

Raster texture:

Vector texture:



Problem

Database: Sequence of paths

Query: given point in 2D space, find frontmost enclosing path and distance(s) to nearest visible curves

Observations:

- Distance used for analytic anisotropic antialiasing
  - Can also be used for special effects, such as embossing and shadows
  - Multiple distances (knn) useful in some circumstances
- Signed distance or winding rule can be used for path enclosure test
  - Winding rule more general: can handle self-intersecting curves
  - Requires counting intersections of line with curves

Indexing and Search

Index: Grid of lists

Algorithm:

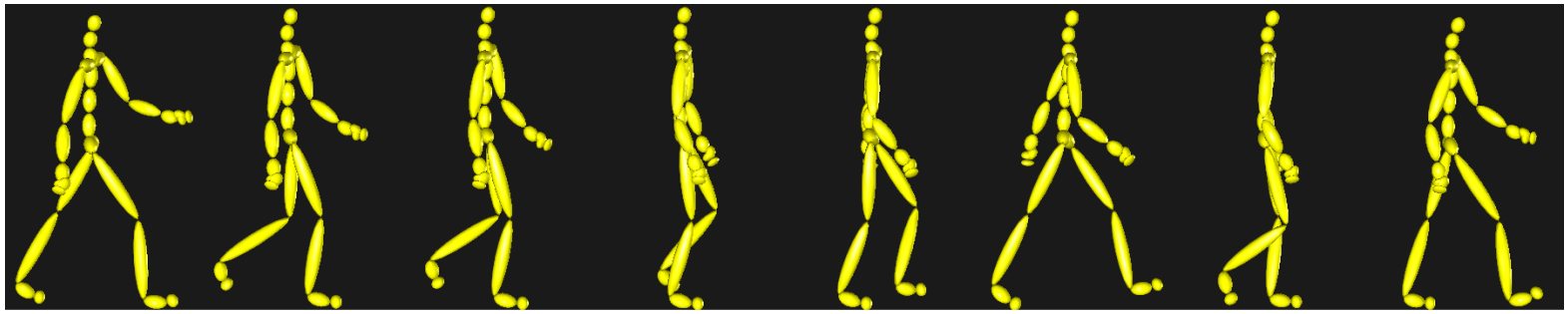
1. Process paths covering point back-to-front
2. Keep closest distances to curves on topmost path
3. Once closest distance(s) found, compute region colours and blend to antialias

Observations:

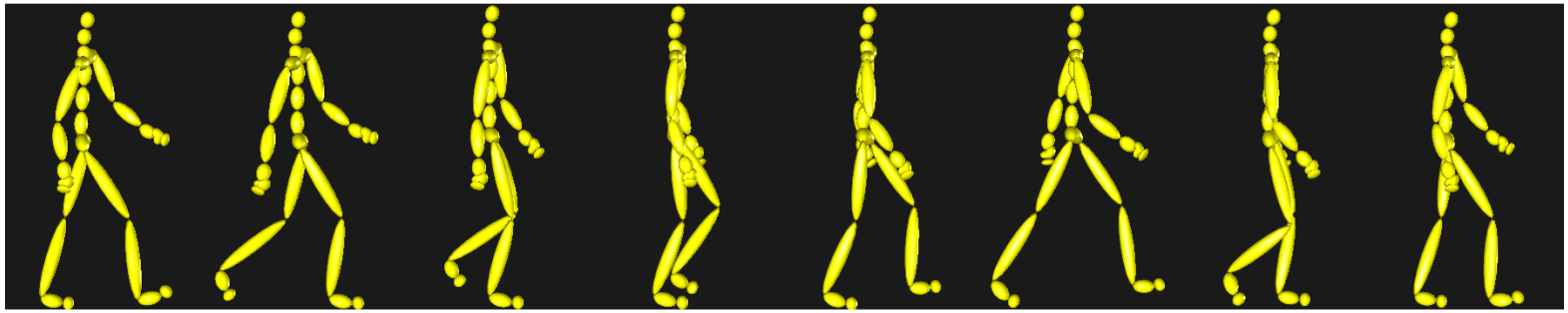
- Efficiency of grid of lists depends on limited density of features
- R-tree/B-tree might also be useful in this case
- Updating not as important if textures static

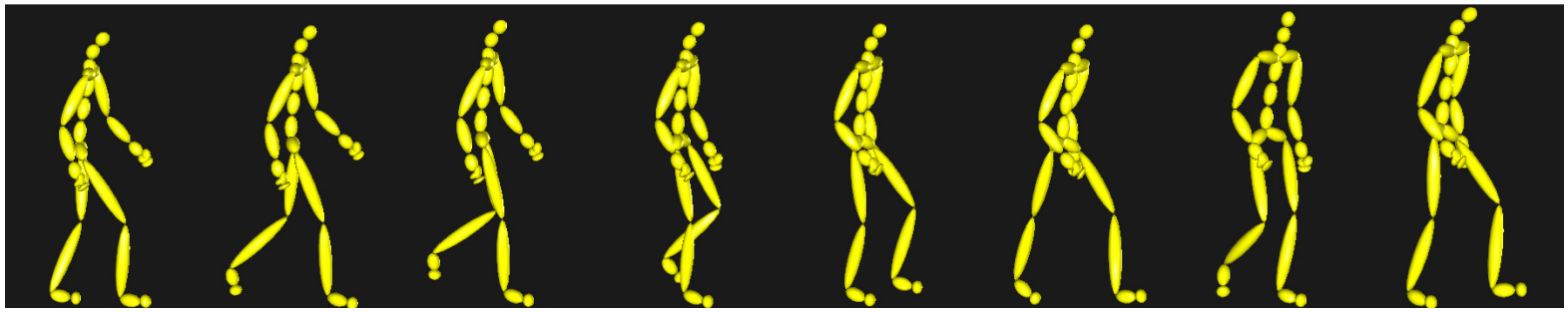
Time Series Search

Query:

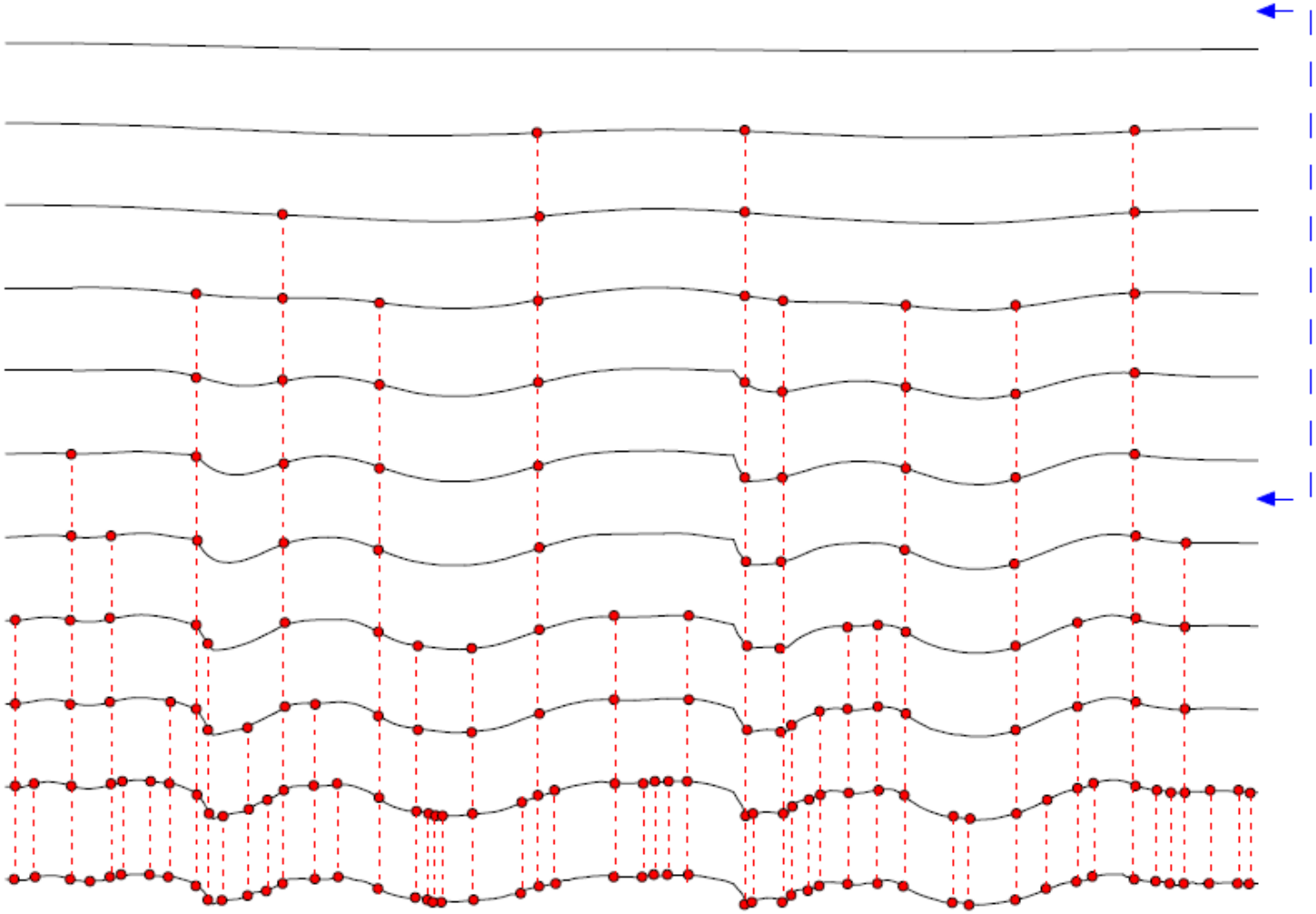


Results:

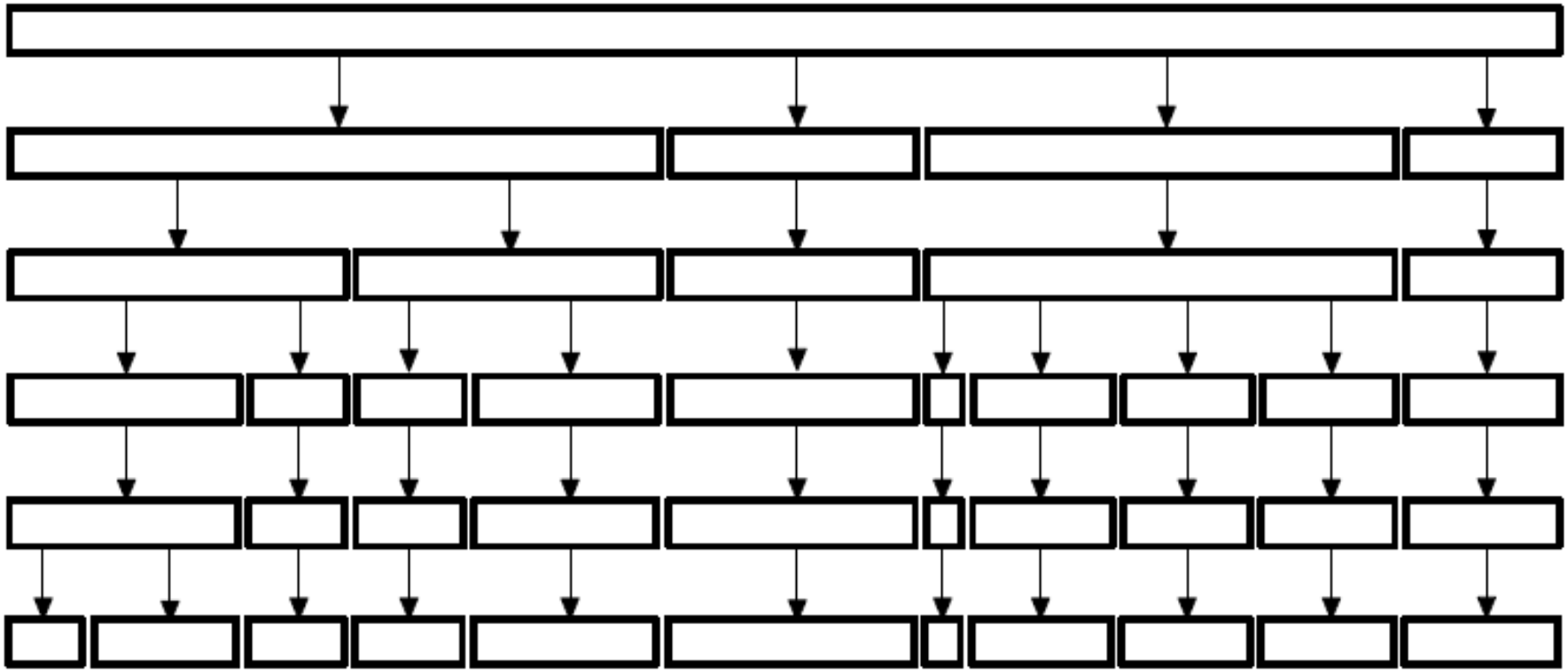




Scale-Space Indexing



Scale-Space Indexing

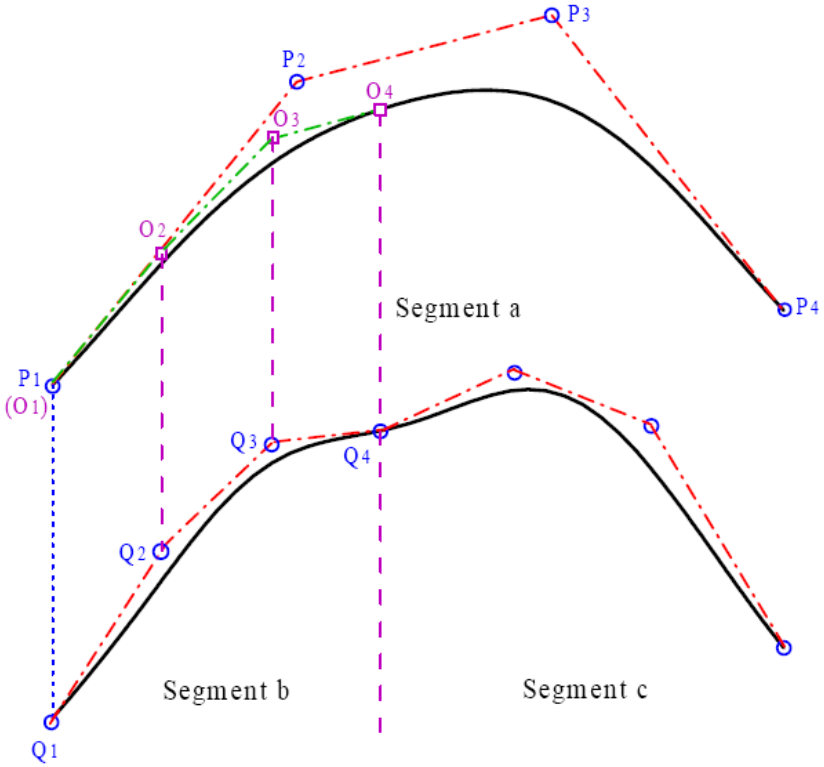


Compression

- Approximate each segment with a spline
- Use spline subdivision to predict across scales
- Code and compress residuals

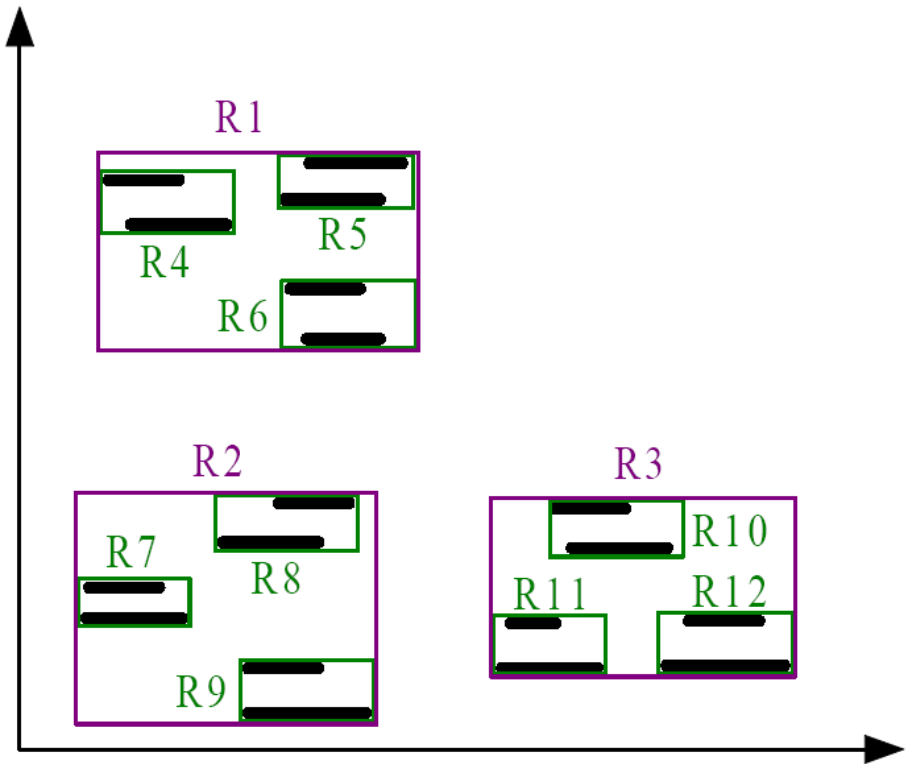
Observations:

- Index organization based on spatial coherence
- Compression based on eliminating redundancy
- *Spatial coherence is a form of redundancy*



Similarity Match and Join

- Envelopes around all spline segments mapped into line segments in n-D space
- Indexed by R-tree
- Similarity match converted into a spatial nearest-neighbour query
- Also permits definition of a “join” operation in terms of spatial R-tree join

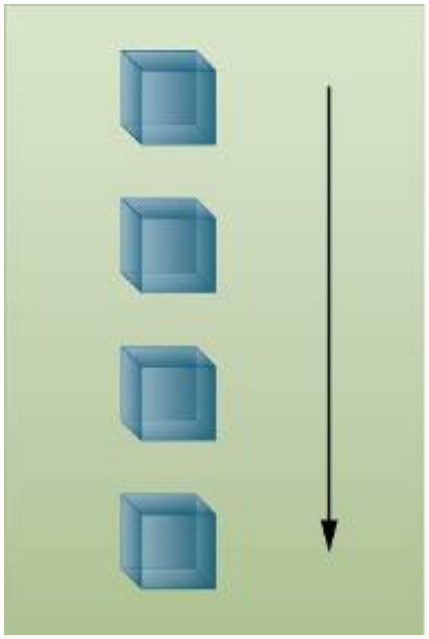


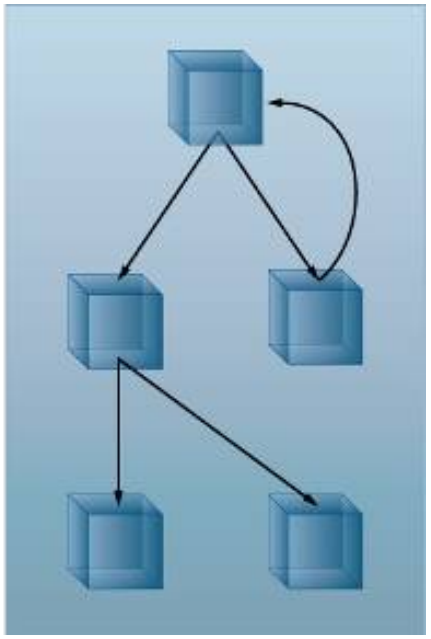
Creating Data Coherence

- Space-filling curve + sorting
  - Convert nD spatial coherence to1D spatial coherence
  - Allows for pointerless balanced structures
  - Easy to update, but not optimal
- Trees
  - Block transfers
  - Good cache alignment and behaviour
  - Support insertion and flexible memory allocation and layout
  - Variable depth → variable latency
- Grids
  - Block transfers
  - Good cache alignment and behaviour
  - Fixed depth → hard to support irregularity

Control Flow
Definitions

- Control flow:
  - Provides ability to avoid work
- Core
  - Unit with independent flow of control
- SIMD Vector
  - All units execute same operations, share control flow
  - Various ways to emulate control flow
  - *Only some "control flow" methods actually avoid work*





Control Flow Emulation

- Often want to emulate control flow on SIMD units
- Modern GPUs use a “SIMD Tile” approach
  - Support many virtual threads (“strands”) in a single tile
  - Use masking to emulate control flow within tile
  - If masks completely empty, can skip operations, otherwise do conditional assignment (BOSCC optimization)
  - Inefficient if control flow divergent within tile
- Alternative approach: packing
  - Support variable-sized tiles
  - Pack active elements together for greater coherence
  - Requires efficient packing/unpacking operation
  - Combine the two: pack only when coherence low
  - Requires dynamic cost/benefit analysis to be effective

Creating Coherence

- Sorting
  - Reorder elements to bring those with similar properties together
  - Need to create 1D key to sort on
- Packing
  - Bring active elements together; discard inactive elements
- Reordering
  - When choices possible, order tasks to maximize temporal locality
- Blocking
  - If bring in block from memory, want to get good use out of rest of block
- Alignment
  - Align data with vector units (e.g. AOS; can compete with blocking)

Conclusions

- Ray tracing is a spatial search problem
- Many related applications
  - Techniques for enhancing performance of ray tracing can also be applied to these problems
  - Already developed techniques for these problems can be applied to ray tracing
- Support for irregular data structures needed
  - Common data structures: Grids and trees
  - Parallel update and caching challenges for each
  - Hardware support would be an interesting avenue to consider
- Exist general techniques for “creating coherence”
  - Sorting, packing, reordering, blocking, alignment
  - Need to do cost/benefit analysis to determine when each applicable