# Visualization of large-scale 3D city models with detailed shadows

Matthias Wagner

tum.3D computer graphics & visualization

# Content

- Introduction
- CityGML
- Data structure
- Rendering
- Shadows
- Ray tracing
- Conclusion and prospects
- Demo

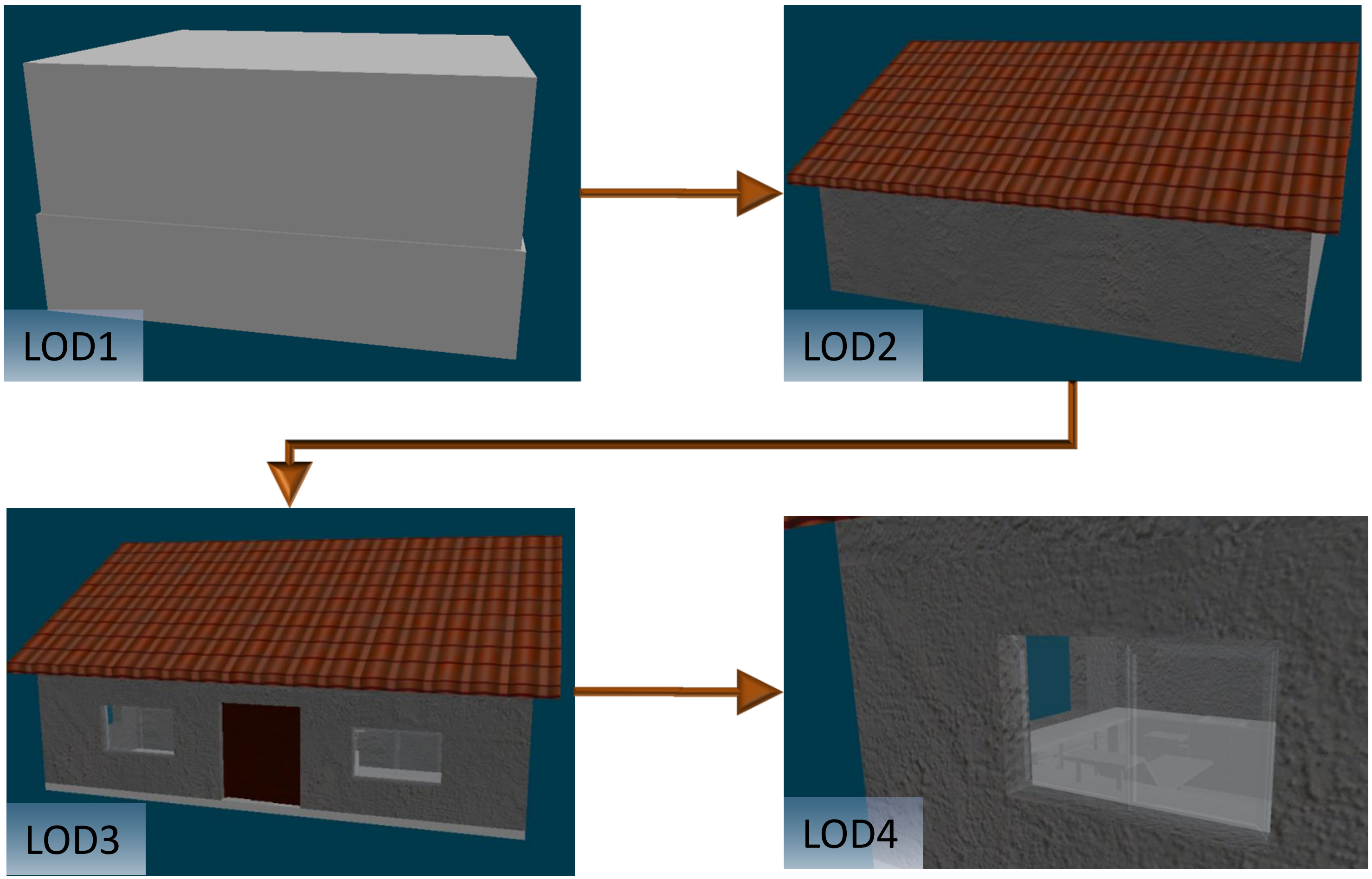
# Introduction

- Thesis objectives:
  - Development of an interactive 3D city model viewer
  - Data source: CityGML
  - Focus on shadow display
  - Support of time-dependent city data
    - Cities developing over centuries
    - No animation

# CityGML

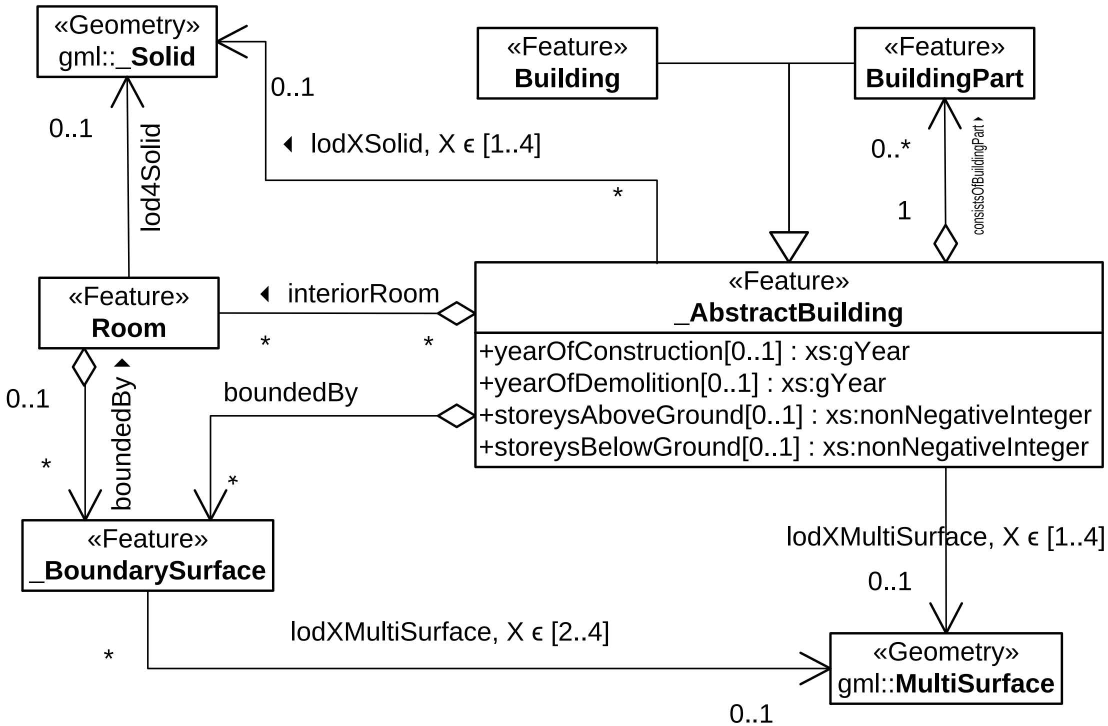
- Information model for representing 3D urban city objects
- Contains different aspects of cities, including:
  – Geometry
  – Appearance
  – Semantics
  – Topography
- Based on the “Geography Markup Language” (GML)

# CityGML



# CityGML

- Building model as example
- Much more semantic information in full model

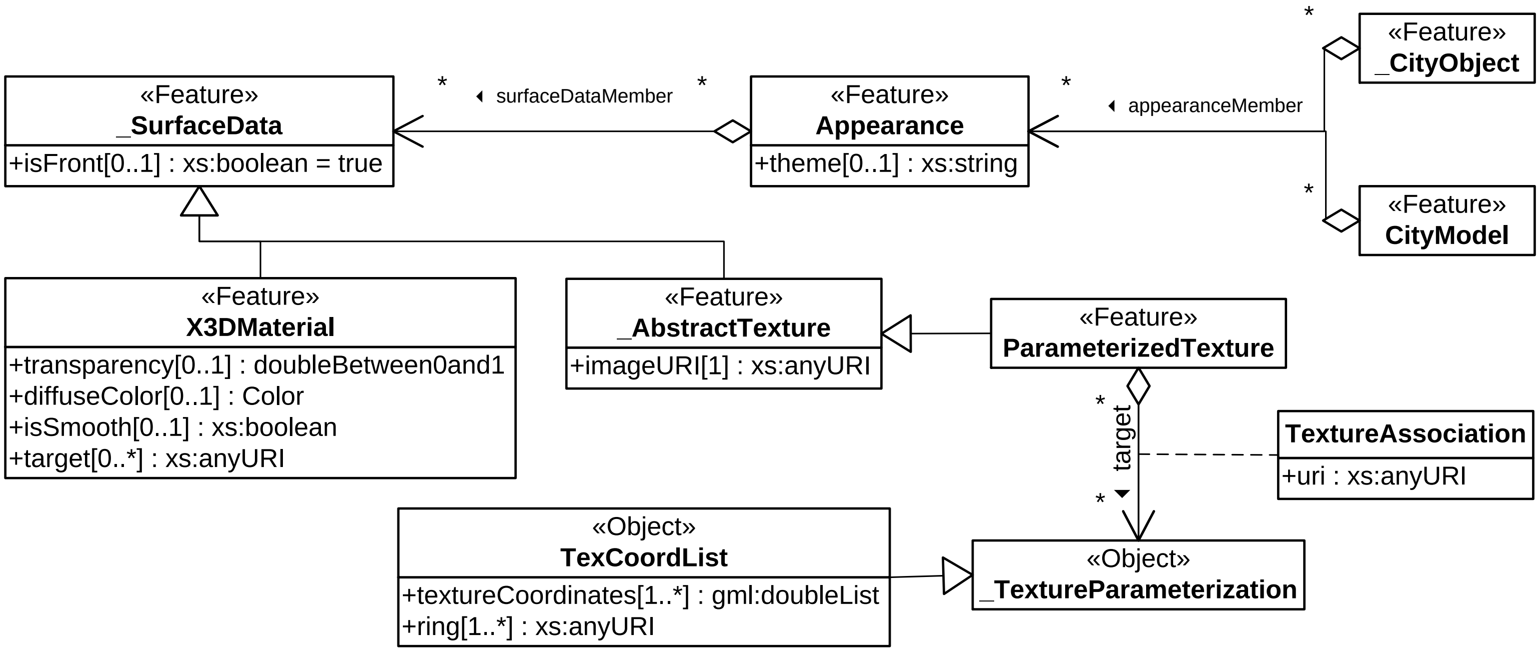


# CityGML

- Appearance model
- Each appearance belongs to a theme (like summer and winter, heat images)
- Appearance defined outside of geometry model
  - Appearances are attached to surfaces
  - Surface itself does not know about its appearance

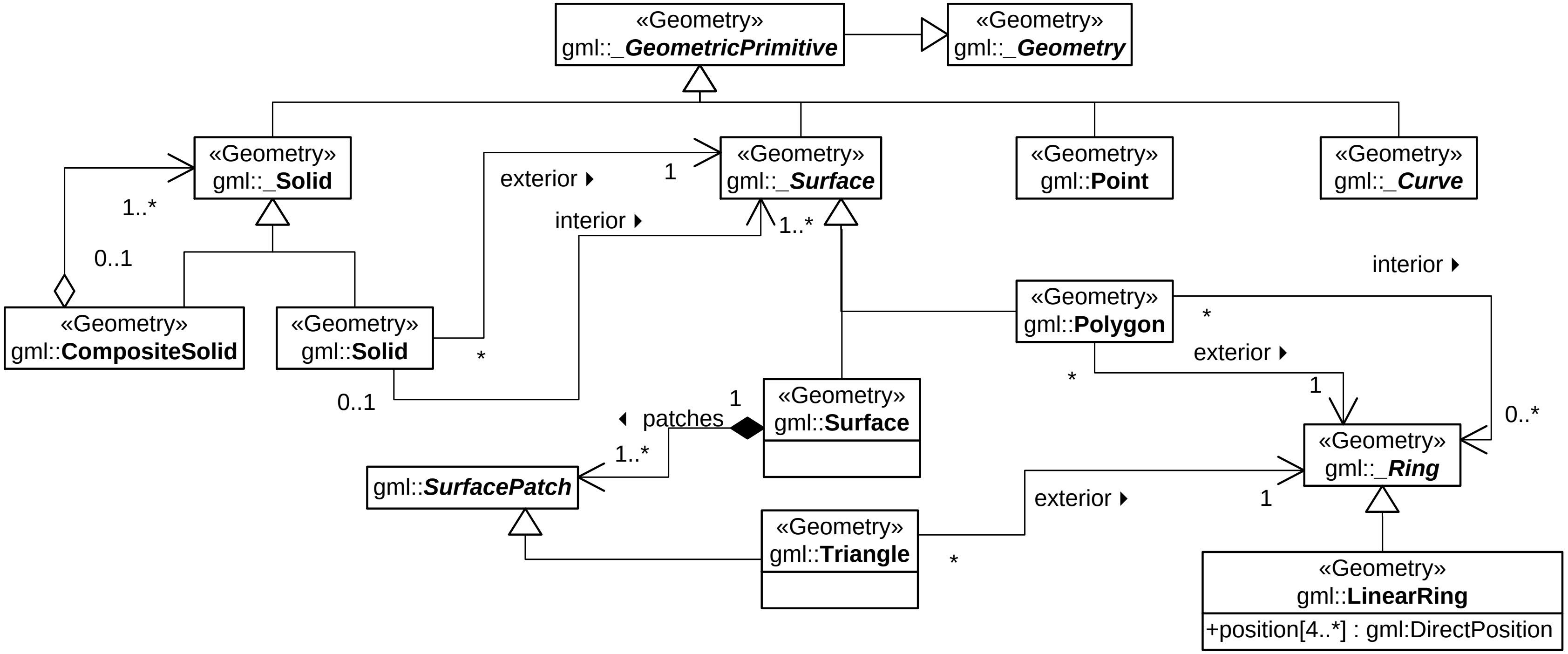
# CityGML

- Appearance model excerpt:



# CityGML

- Uses subset of GML geometry model
- Excerpt:



# Content

- Introduction
- CityGML
- Data structure
- Rendering
- Shadows
- Ray tracing
- Conclusion and prospects
- Demo

# Data structure

- CityGML scene graph inefficient for rendering
  - Many very small draw calls
  - Frequent state changes
  - No usage of occlusion information
- Better: clustering based on
  - appearance to reduce state changes and draw calls
  - spatial coherence to use occlusion information

# Conclusion and prospects

## Geometry data

### CityGML

- Positions (double)
- Texture coordinates

### Application

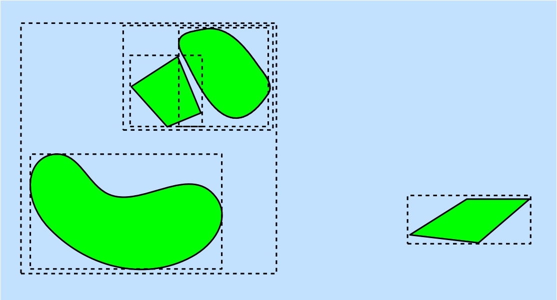
- Positions (float)
- Normals
- Texture coordinates
- Triangulated
- Stored in kd-tree

### GPU

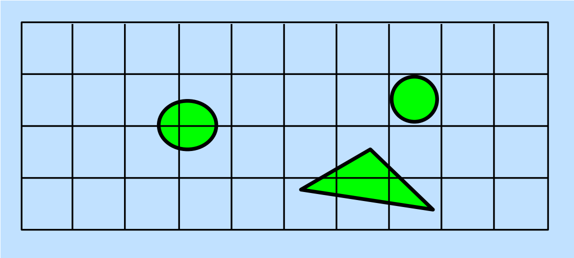
- Similar to application
- Compressed!

# Data structure

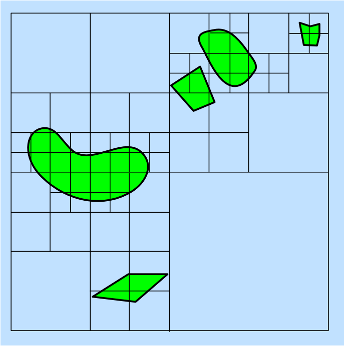
- Spatial coherence
  - Bottom-up
    - Bounding volume hierarchies



  - Top-down (spatial subdivision)
    - Uniform



    - Octree
    - BSP tree
      - Kd-tree (axis aligned BSP)



# Data structure

- Kd-tree chosen because of
  - Performance
    - For rendering and ray tracing
  - Simplicity
    - Axis aligned splitting planes allow many simplifications
  - Well-known
  - Flexibility
    - Incorporation of time dimension

# Data structure

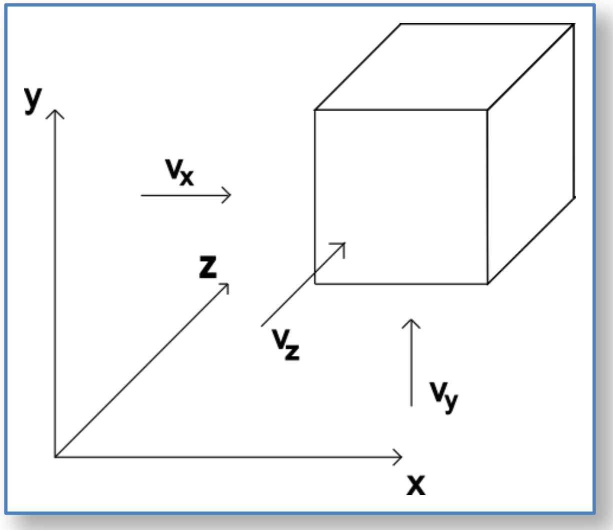
- Kd-tree construction
  a) Insert all objects at once
  b) Choose splitting plane
     1) Split dimension
     2) Split value
  c) Divide objects into left and right and recursively continue with a) until termination criterion is met
     1) Split intersecting objects

# Data structure

- How to choose splitting plane?
  - Naïve:
    - Current depth specifies split dimension
    - Split at center of split dimension
  - Cube-like voxels
    - Split dimension has largest extent
    - Split at center
  - Binary search
    - Split dimension has largest variance
    - Split at median of split dimension

# Data structure

- Construction optimized for culling empty space
  - Surface Area Heuristic (SAH) as cost prediction function of a split
  - SAH idea is based on assumptions
    - Rays (or view directions) distributed equally through space
    - Rays cannot be blocked by scene objects




  - Surface of kd-tree node used to approximate ray intersection probability

# Data structure

- Probability of ray intersecting $V_L$ and $V_R$, given V is hit:

$$P(V_L | V) = \frac{SA(V_L)}{SA(V)} \qquad P(V_R | V) = \frac{SA(V_R)}{SA(V)}$$

- Surface area:

$$SA(V) = 2(V_{width}V_{depth} + V_{width}V_{height} + V_{depth}V_{height})$$

- Cost of a split ($N_L$, $N_R$ give polygon count):

$$Cost_{split}(V_L, N_L, V_R, N_R) = C_{traversal} + C_{intersection}(P(V_L | V)N_L + P(V_R | V)N_R)$$

# Data structure

- How can SAH be used for 4D data?
  - "Surface area" of a "voxel" is now a volume
  - Straightforward:

$$SA(V) = 2(V_{width}V_{height}V_{depth} + V_{width}V_{height}V_{duration} + V_{width}V_{depth}V_{duration} + V_{height}V_{depth}V_{duration})$$

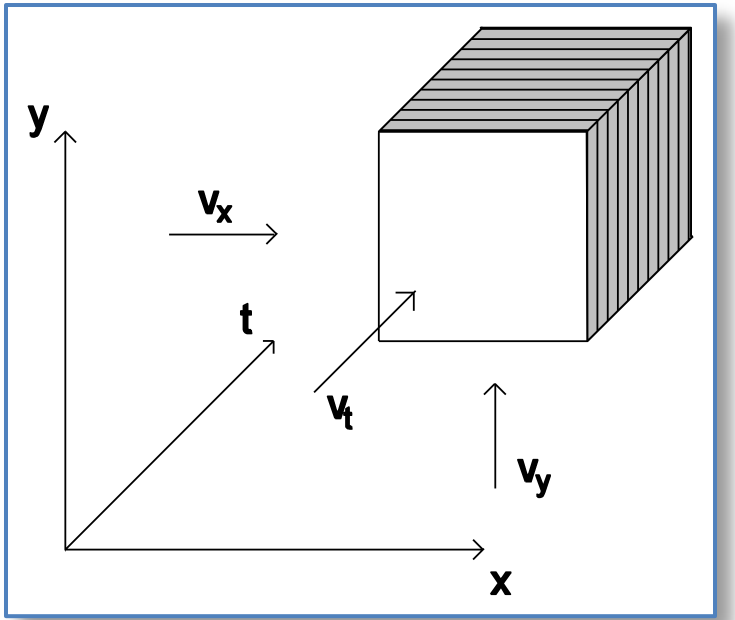
  - This assumes rays to be distributed evenly in 4D-space
  - However, we only display one point in time, which does not fulfill this assumption

# Data structure

- Rays have constant time coordinate, resulting in

$$SA(V) = 2(V_{width}V_{height}V_{duration} + V_{width}V_{depth}V_{duration} + V_{height}V_{depth}V_{duration})$$

  - Duration of 1 ⇒ 3D case
  - Only shaded areas (and opposite area) included



# Data structure

- Kd-tree construction is expensive
- But: whole scene known
  - No animations
- Possible future extension ideas:
  - Animations with small geometry influence
    - Water waves
  - Transformed objects
    - Transform local kd-tree of object
  - Real-time construction for complex animations

# Content

# Rendering

- Efficient rendering
  - Vertex sharing
    - Should not destroy coherency of indexed vertices
  - Draw call concatenation
  - Few state changes
- Lead to introduction of *draw batches*
  - Same material, texture and shader
  - Constructed from kd-tree leaf objects
  - Sorted and merged (inside a leaf)

# Rendering

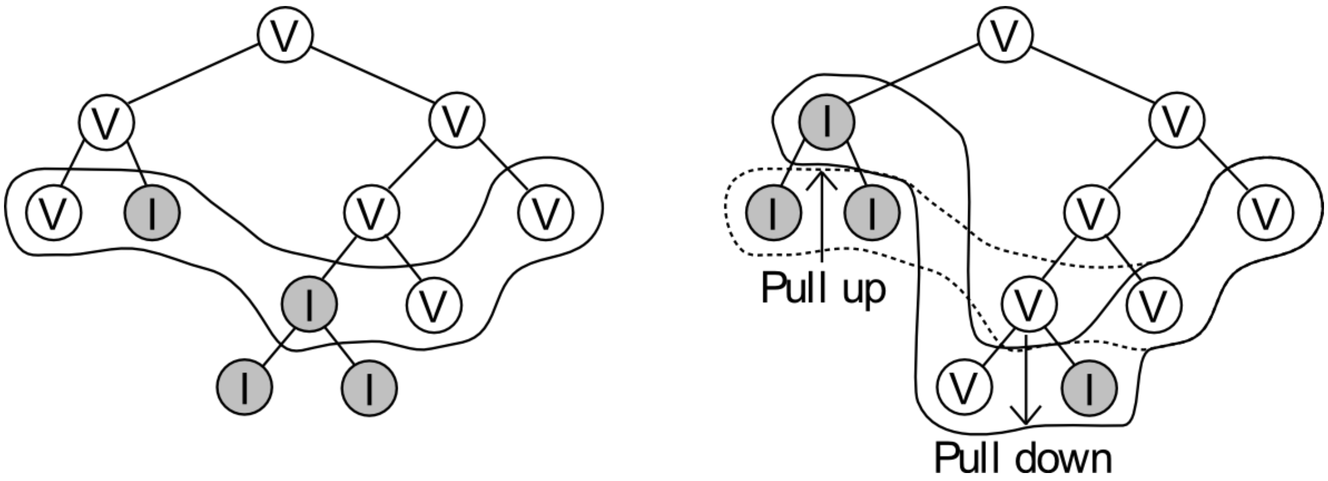
- Kd-tree rendered front-to-back using draw batches
- Frustum culling applied
- Exploiting occlusion queries
  - Hierarchical stop-and-wait? – No!
  - Instead: *Coherent Hierarchical Culling*!

# Rendering

- *Coherent Hierarchical Culling* exploits
  - Temporal coherency
    - Visible nodes assumed to stay visible
      - Interior nodes: direct traversal
      - Leaves: occlusion query of bounding box directly followed by rendering node geometry (no waiting)
    - Invisible nodes always wait for query result
  - Interleaving
    - Queries stored in queue, handled when traversal stack empty or front query finished

# Rendering

- Visibility stored using boolean and last-visited frame ID
  - Nodes initialized to not visible
  - Visible  nodes mark parents as visible when queries return



# Content

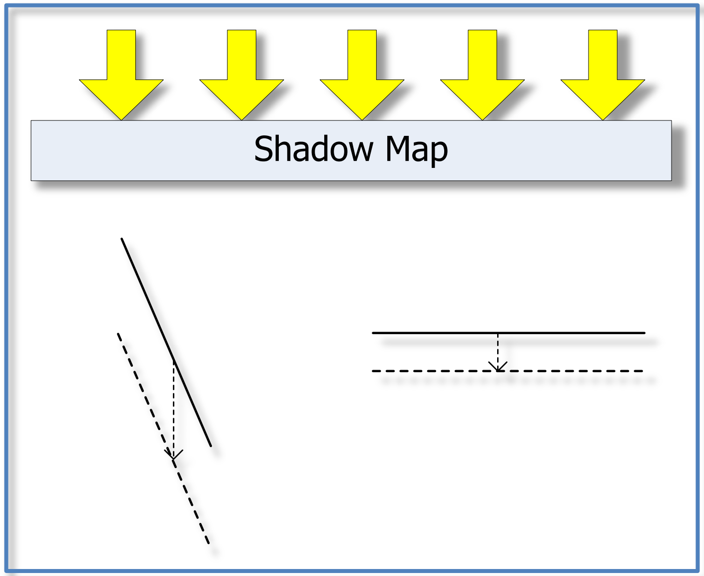
- Introduction
- CityGML
- Data structure
- Rendering
- Shadows
- Ray tracing
- Conclusion and prospects
- Demo

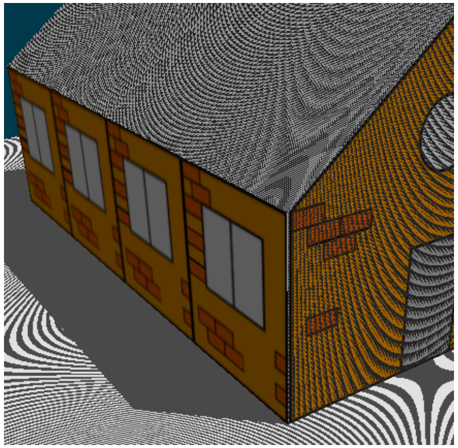
# Shadows

- Raycasting
  - Very high quality…
  - … but very (?) slow
- Shadow volumes
  - High quality
  - Doesn‘t scale well with scene complexity
- Shadow maps
  - Quality depends on scene extent/complexity
  - Fast

# Shadows

- Shadow maps have to deal with
  - Self-shadowing
    - "Fixed" by adding a bias
      - Depth bias
      - Slope scaled depth bias
      - Clamped to max depth bias



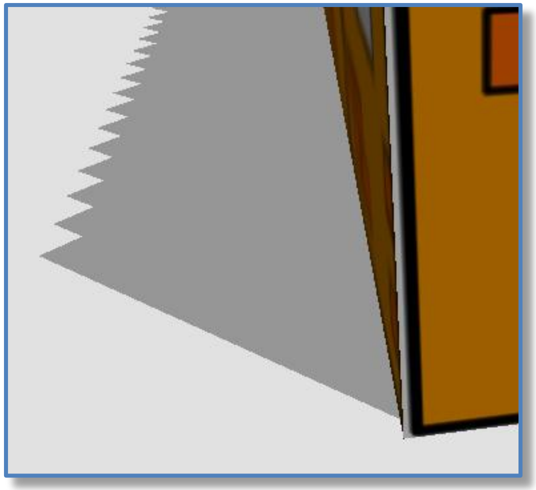



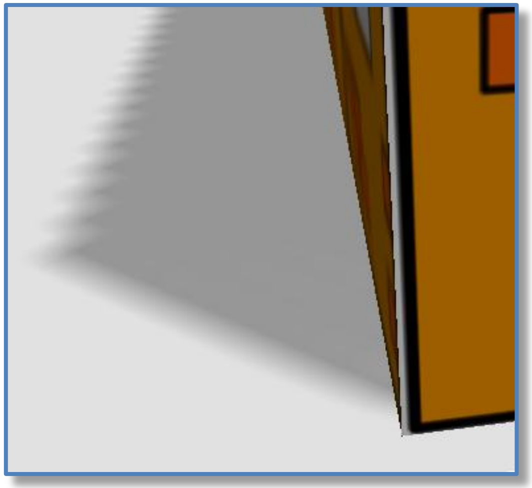
# Shadows

- Shadow maps have to deal with
  - Projection aliasing
    - Caused by different view angles
    - Hard to fix
  - Perspective aliasing
    - Caused by perspective view of the camera
      - Closer objects bigger on screen
    - Scales projection aliasing error
    - Several methods to reduce perspective aliasing

# Shadows

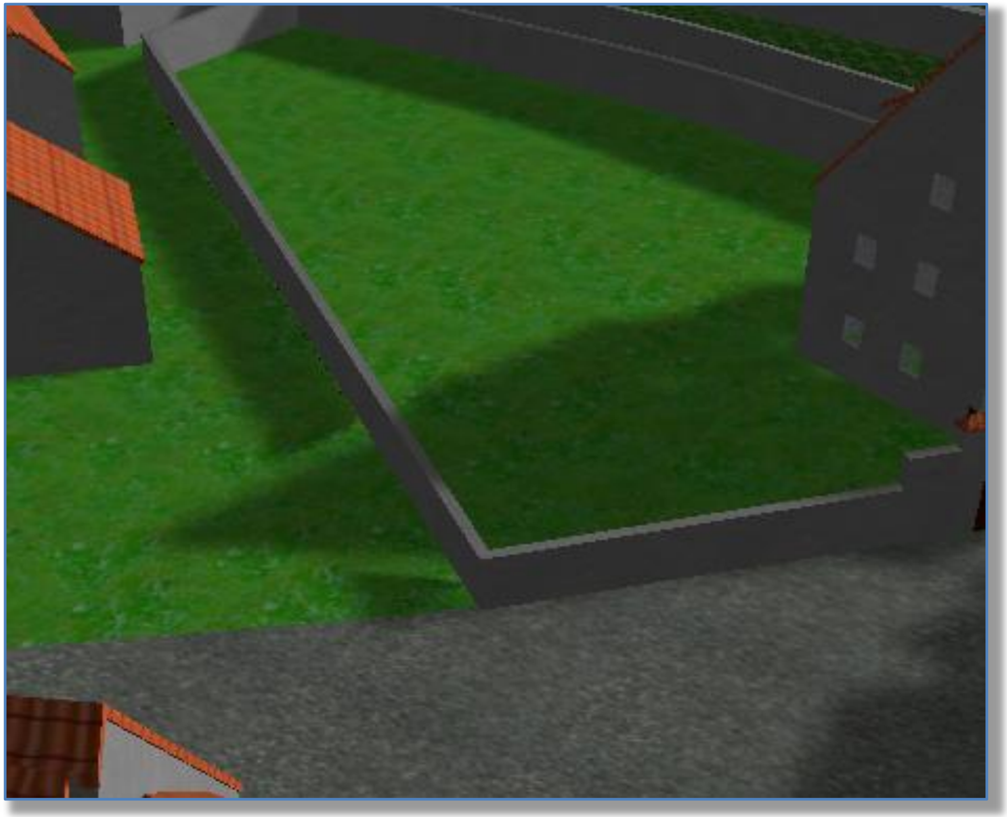
- Smoothing shadows
  - Variance shadow maps
    - Store mean and squared mean of a depth distribution
    - May be filtered using standard techniques
    - Allows to calculate mean and variance





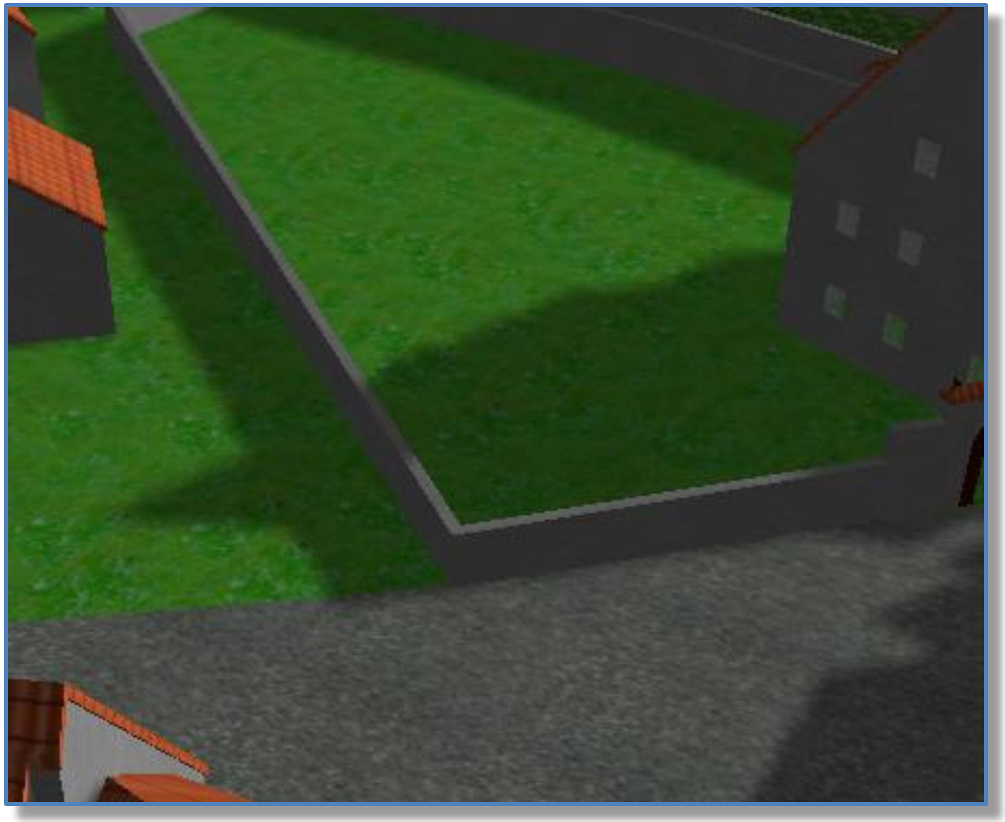
# Shadows

- Variance shadow maps
  - Light bleeding



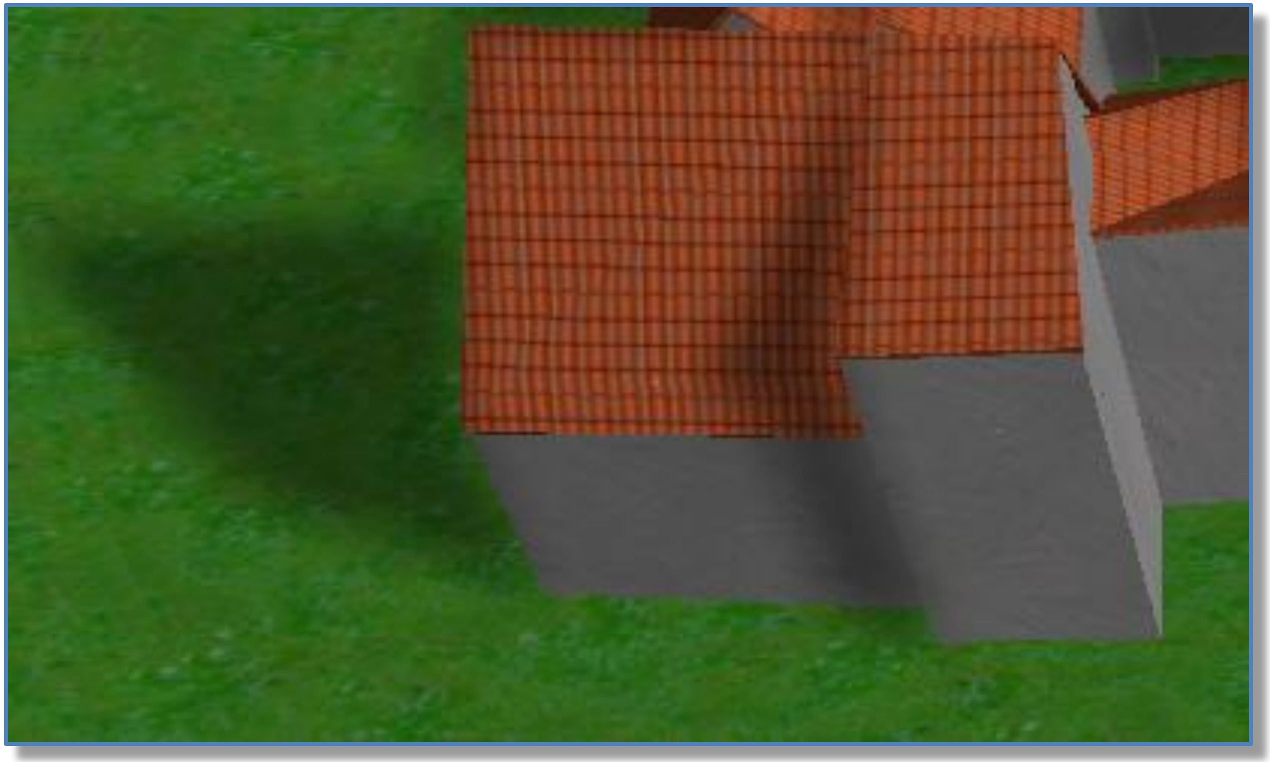
# Shadows

- Screen space Gauss smoothing
  - Large filter size
  - Convolution of Nx1 and 1xN kernel
  - Smooth strength based on distance



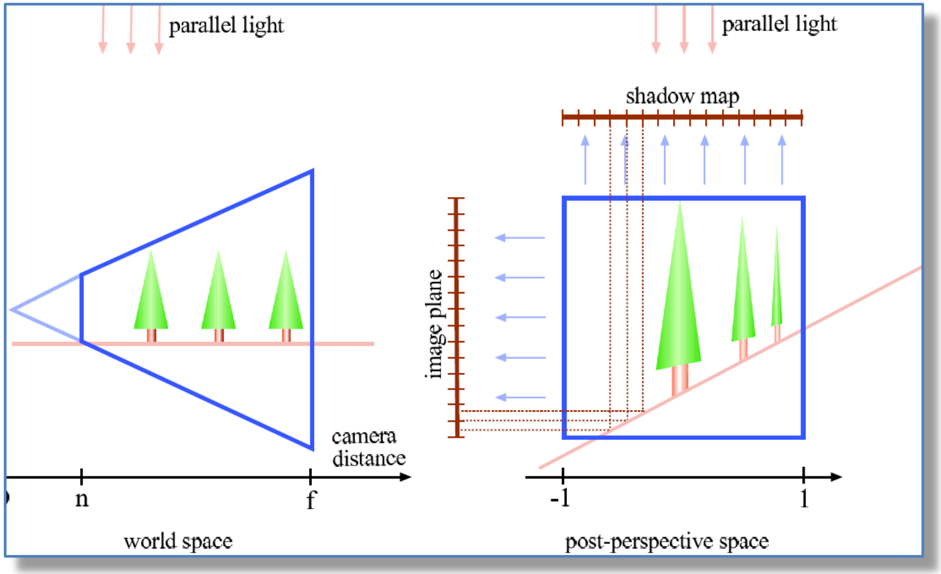
# Shadows

- Screen space smoothing needs to consider depth buffer information to avoid wrong smoothing



# Shadows

- Perspective Shadow Maps
  - Reduce perspective aliasing
  - Increase resolution of shadow map close to camera
  - Done in post-perspective space

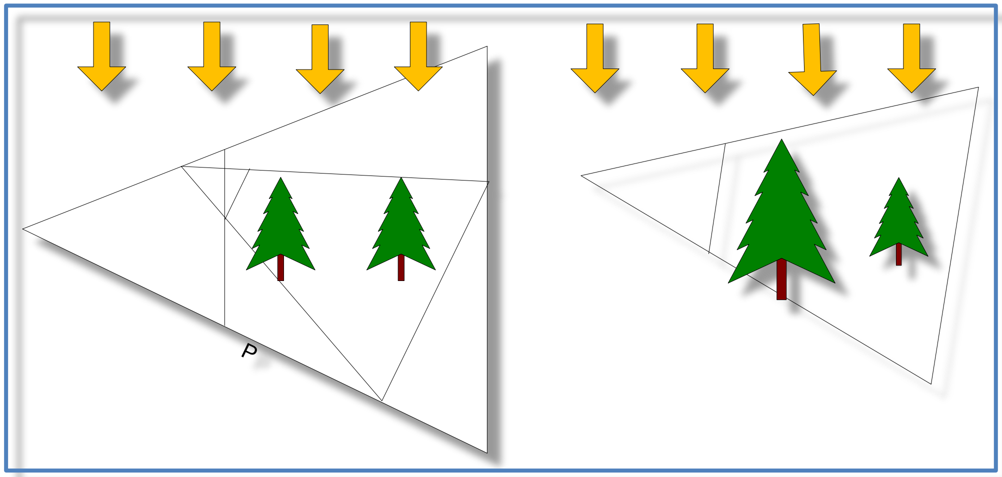


# Shadows

- Perspective Shadow Maps
  - Have to deal with implementation issues
    - Light sources may change type post-perspective
    - Objects behind camera on infinity plane

# Shadows

- Light space perspective shadow maps
  - Any projection transformation can warp the shadow map
  - Warp must only affect shadow map plane

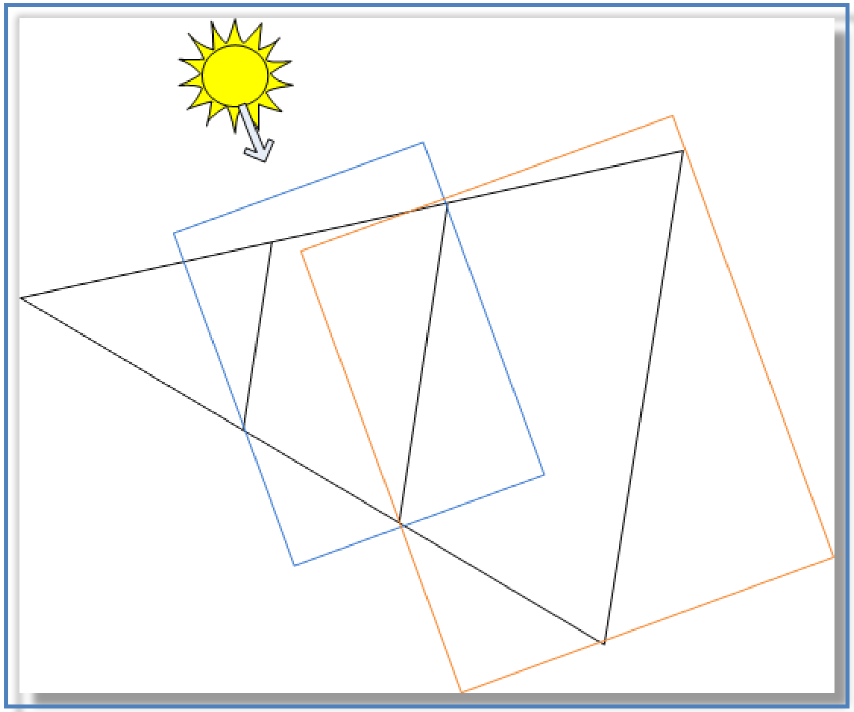


# Shadows

- Extended Perspective Shadow Maps
  - Two steps
    - Find optimal warping effect
      - Warp direction = camera view projected on shadow map plane
    - Find affine transformation not affecting this warping

# Shadows

- Cascaded Shadow Maps
  - Several shadow maps
  - Focusing on different parts of view frustum
  - Implemented using texture arrays
  - Can be combined with previous ideas



# Shadows

- Shadow map approaches still suffer from aliasing problems
- Idea: combine shadow maps and ray tracing
  - Use shadow maps for rough estimation
  - Ray trace shadow boundaries
  - Write primitive ID and shadow flag to texture
  - Create “refine image” based on this texture

# Shadows

- How to find shadow edges?
  - Currently using variance shadow maps
  - Does work quite well, but artifacts occur
  - Other approaches should be researched

# Content

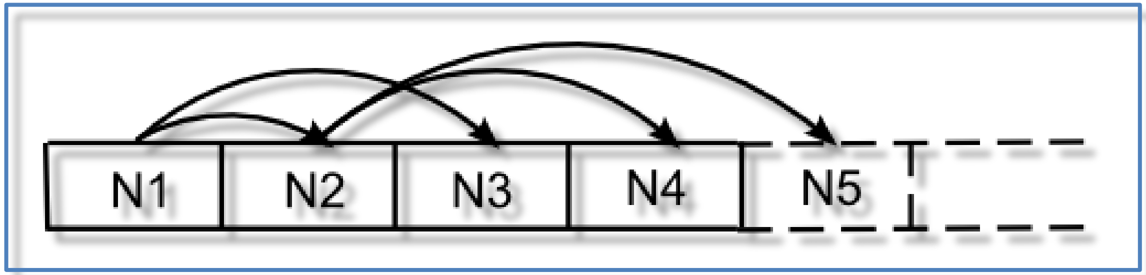
- Introduction
- CityGML
- Data structure
- Rendering
- Shadows
- Ray tracing
- Conclusion and prospects
- Demo

# Ray tracing

- Efficient implementation necessary for interactive frame rates
- Ray-triangle intersection
  - Intersection of ray-plane
  - Projection of triangle to 2D in order to find barycentric coordinates
  - Precomputation of values
    - Cache efficient by avoiding indices and vertices

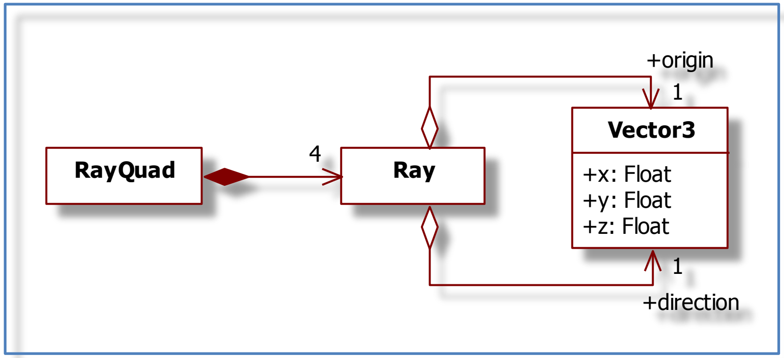
# Ray tracing

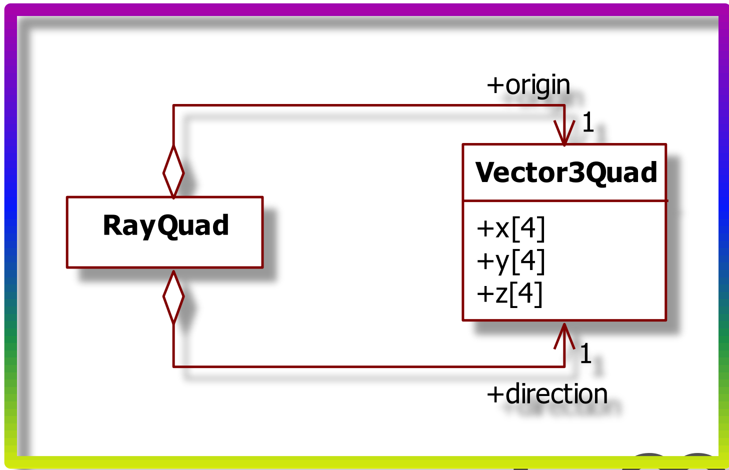
- Kd-tree
  - Traversal algorithm based on ray segments (1D)
  - Front-to-back ⟹ Early ray termination
  - Cache efficient "flat" kd-tree with 8 bytes/node



# Ray tracing

- Multithreading
  - Scales very well, each thread handling a “brick”
- SSE
  - Apply one instruction on multiple data (four floats)
  - Best parallelism by structure of arrays



# Ray tracing

- SSE kd-tree traversal
  - Four rays at once ⇨ traversal order must be unique!
    - Either same origin or same direction signs
- Shadow rays
  - Shadow casters can be cached
    - No need to find the closest intersection!
  - One cache list for each brick

# Content

- Introduction
- CityGML
- Data structure
- Rendering
- Shadows
- Ray tracing
- Conclusion and prospects
- Demo

# Conclusion and prospects

- Kd-tree fits well to time-dependent data
- Hybrid shadow approach feasible
- Improvements
  - Tiling and caching of large cities
  - GPU ray tracing
  - Better shadow edge detection
  - Editing functionality
  - Animations

# Content

# Sources

- These slides have originally been created for the colloquium of the diploma thesis. Figures and models from the following sources have been used (both directly and modified):
  - http://www.citygml.org/
  - http://www.cg.tuwien.ac.at/research/vr/lispsm/
  - http://doi.acm.org/10.1145/566570.566616
  - http://portal.opengeospatial.org/files/?artifact_id=22120