Practical Grammar-based Procedural Modeling of Architecture

SIGGRAPH Asia 2015 Course Notes

Michael Schwarz
Esri R&D Center Zurich (*formerly*)

Peter Wonka
KAUST

Abstract

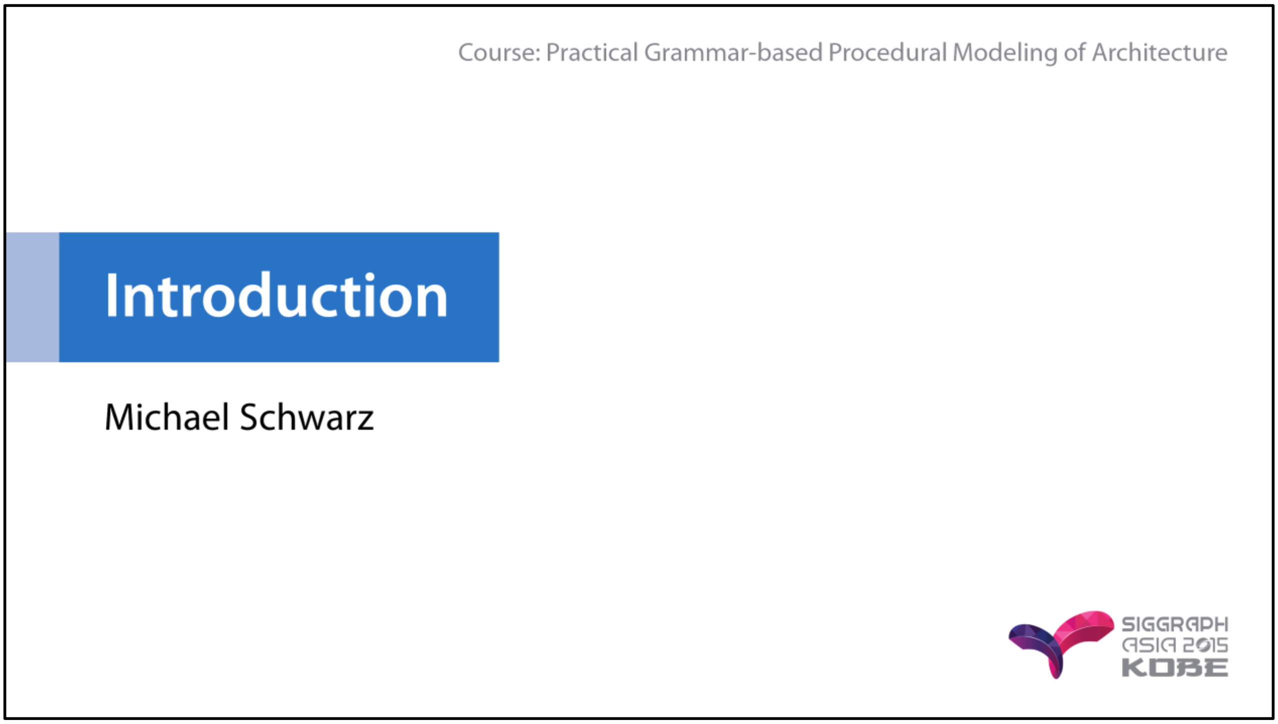
This course provides a comprehensive, in-depth introduction to procedural modeling of architecture using grammar-based approaches. It first presents all necessary fundamentals and discusses the various advanced features of grammar languages in detail. Subsequently, context sensitivity, which is crucial for many practical tasks, and the different forms of support for it are addressed extensively. The course concludes by looking into several further advanced aspects, such as local edits or GPU-based variants.

Elements from a large body of work are covered and presented in a coherent, structured way. The course explores the range of solution approaches, provides examples, and identifies limitations; it also highlights and investigates practical problem cases.

The course is useful for practitioners and researchers from many different domains, ranging from urban planning, geographic information systems (GIS) and virtual maps to movies and computer games, with interests ranging from content creation to grammar-based procedural approaches in general. They learn about the arsenal of available techniques and obtain an overview of the field, including more recent developments. The audience benefits from a coherent treatment of ideas, concepts, and techniques scattered across many (sometimes lesser-known) publications and systems. This course helps in developing a realistic understanding of what can be done with current solutions, how difficult and practical that is, and with which tasks existing approaches cannot cope.

Contents

1 Introduction



Course: Practical Grammar-based Procedural Modeling of Architecture

Introduction

Michael Schwarz

Procedural modeling

Model objects by specifying a procedure of how to construct/generate them

Different approaches/kind of procedures for different objects

This course:

Grammar-based approaches

- Grammar = set of rules + …
- Principle: successive refinement guided by these rules

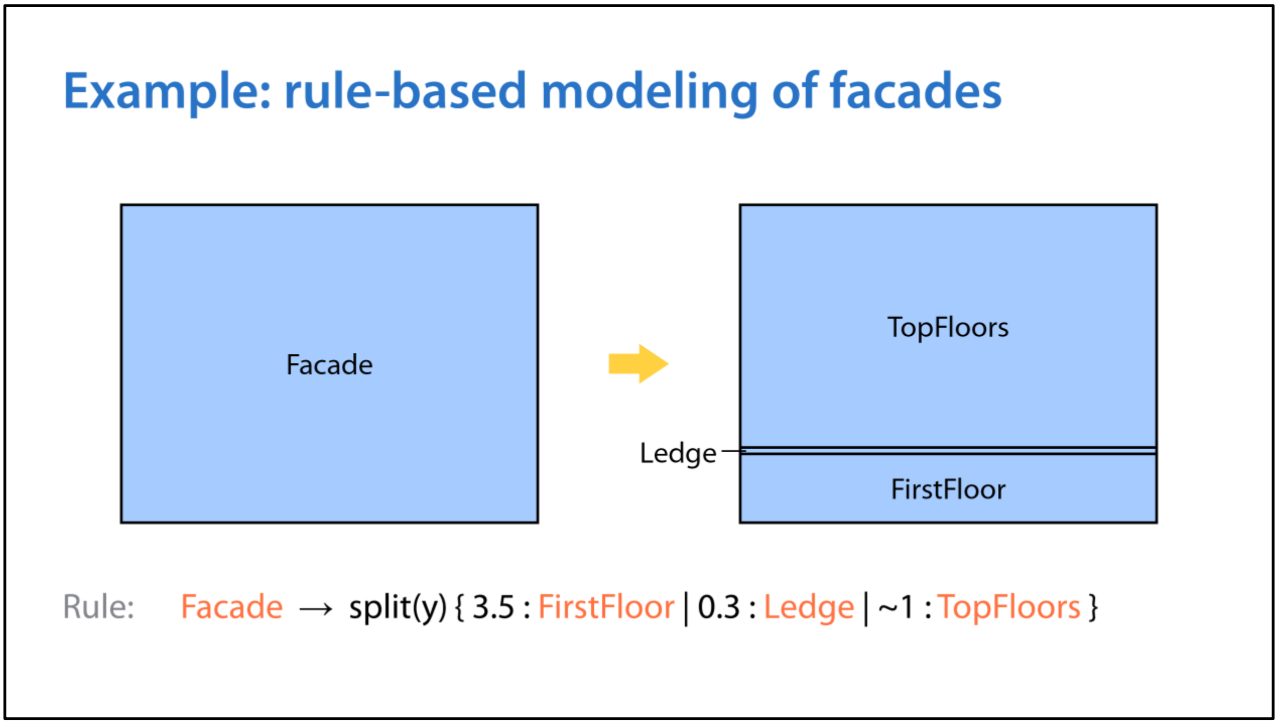
Shapes

- Primarily man-made structures encountered in architecture

Example: rule-based modeling of facades

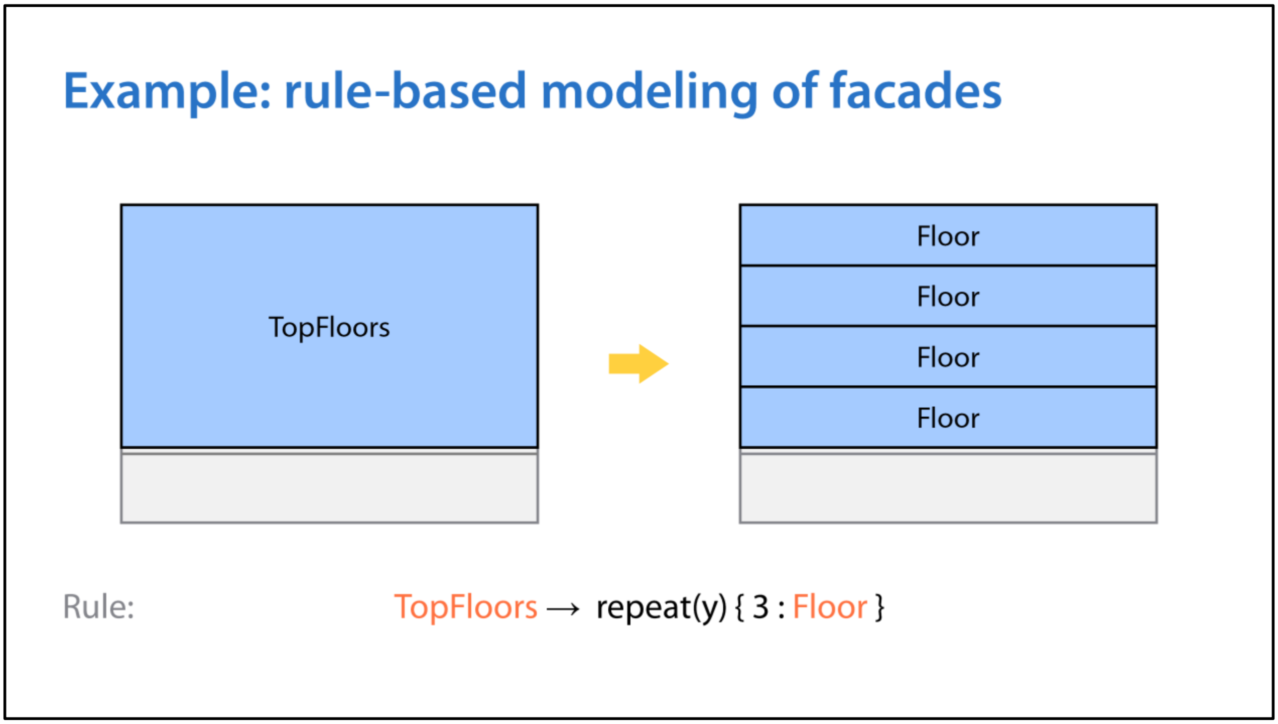
Real-world facade

Vertical structure



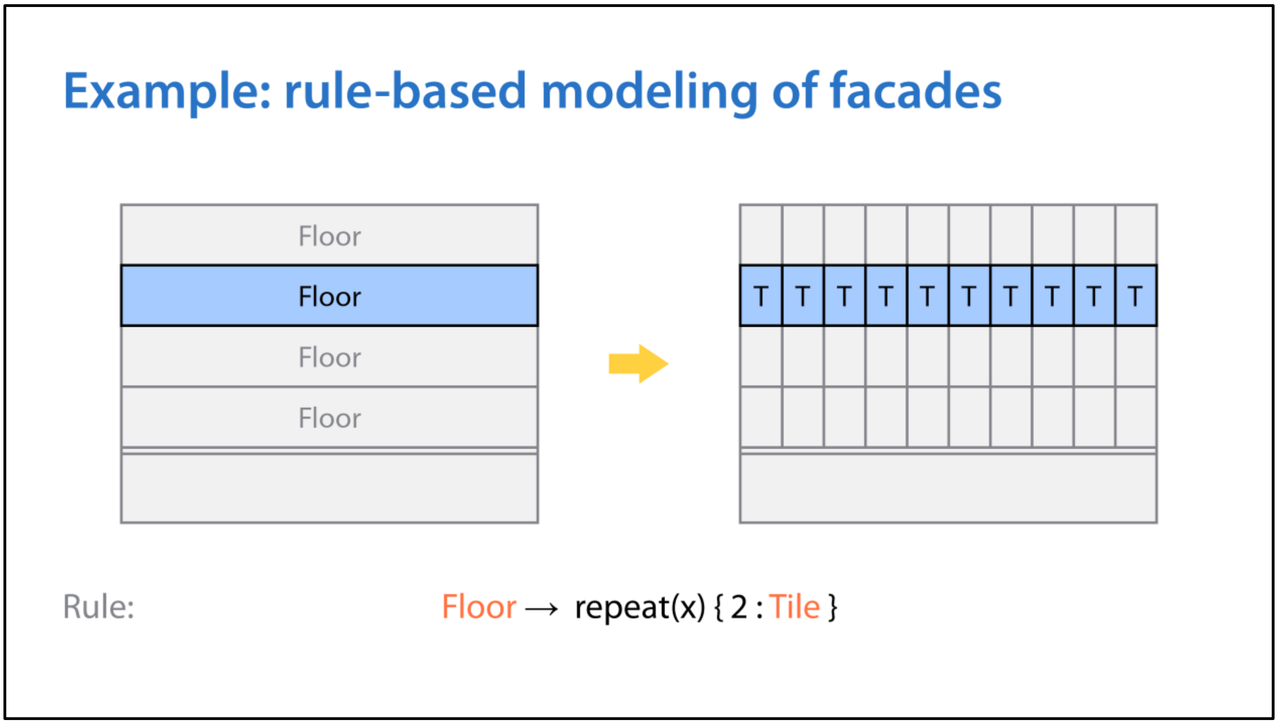
Example: rule-based modeling of facades

Rule: Facade → split(y) { 3.5 : FirstFloor | 0.3 : Ledge | ~1 : TopFloors }



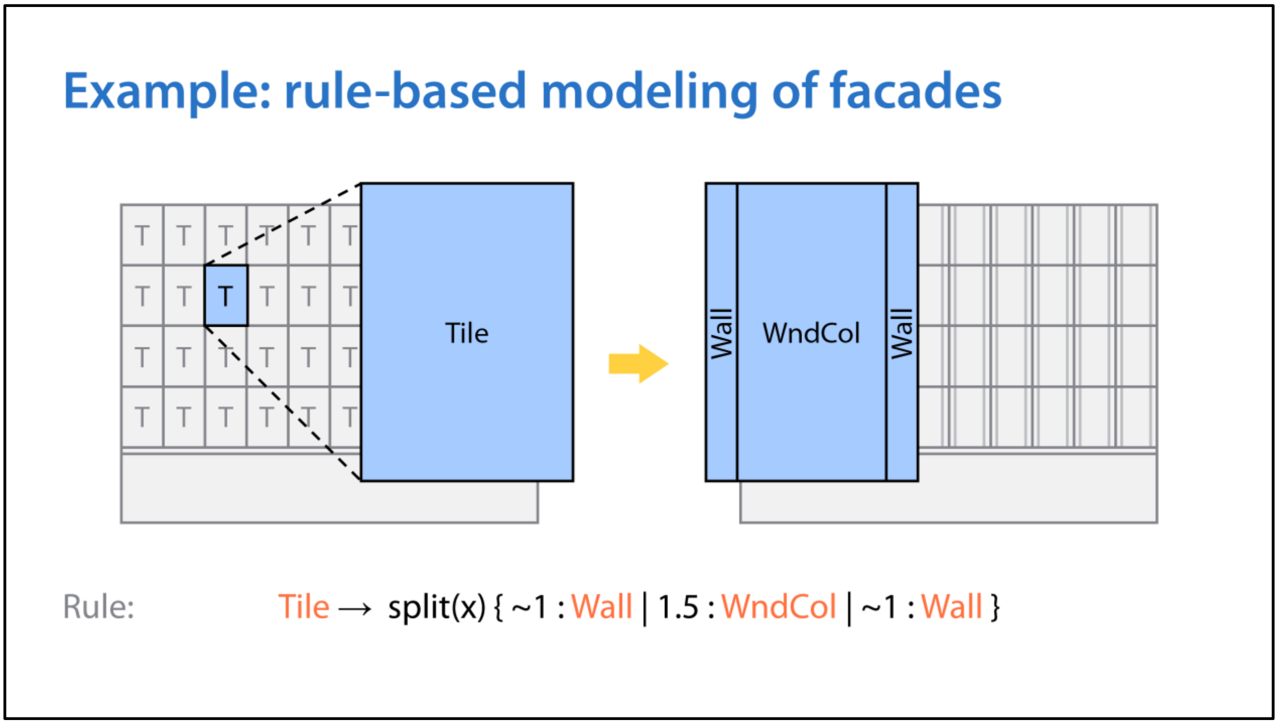
Example: rule-based modeling of facades

Rule: TopFloors → repeat(y) { 3 : Floor }



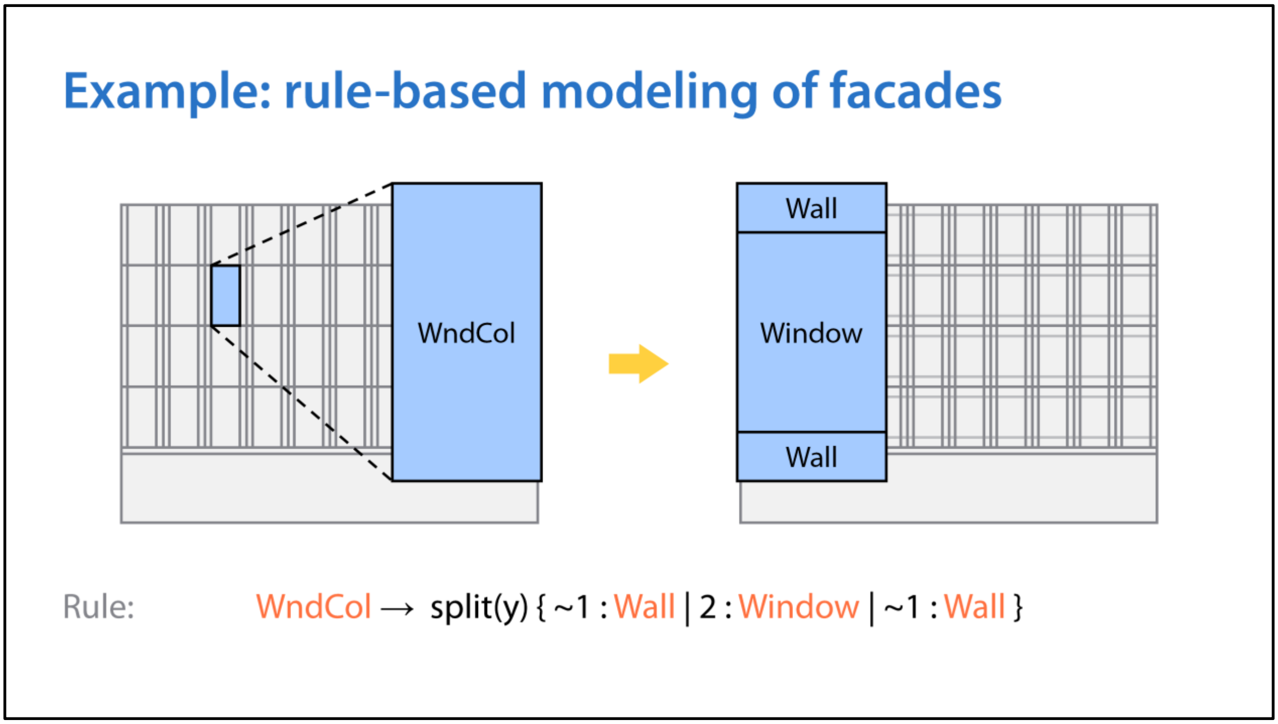
Example: rule-based modeling of facades

Rule: Floor → repeat(x) { 2 : Tile }



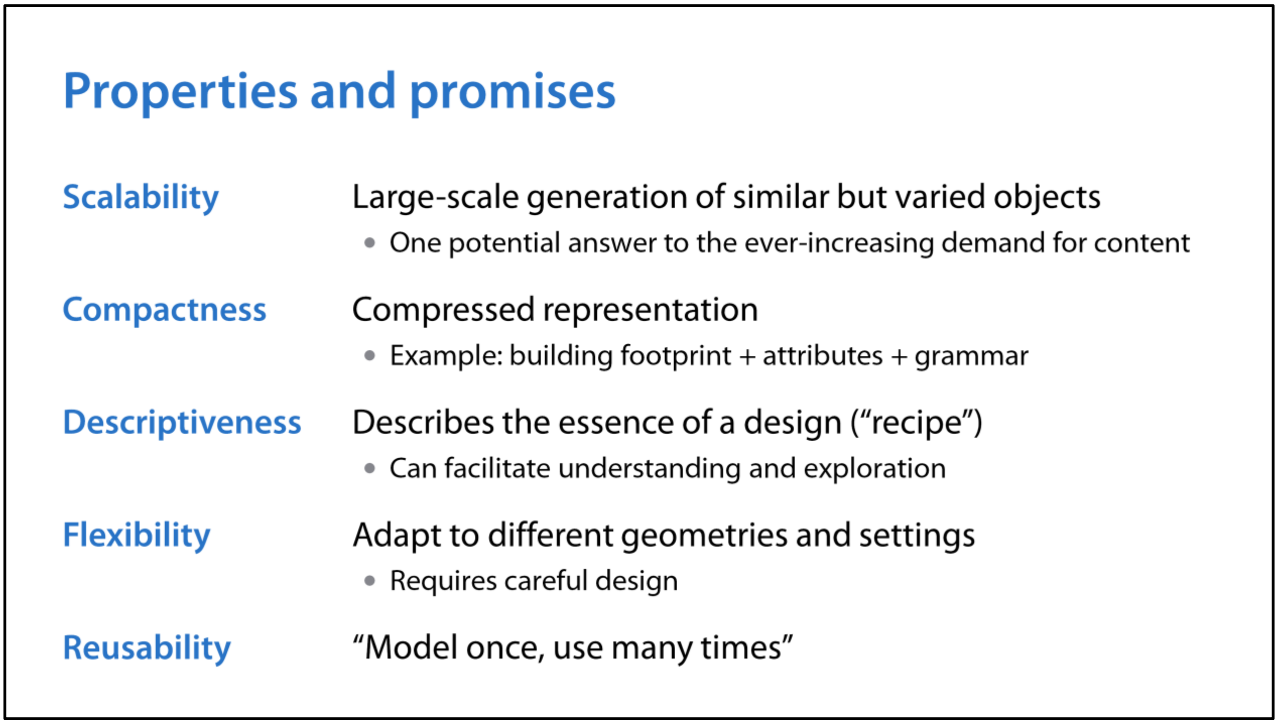
Example: rule-based modeling of facades

Rule: Tile → split(x) { ~1 : Wall | 1.5 : WndCol | ~1 : Wall }



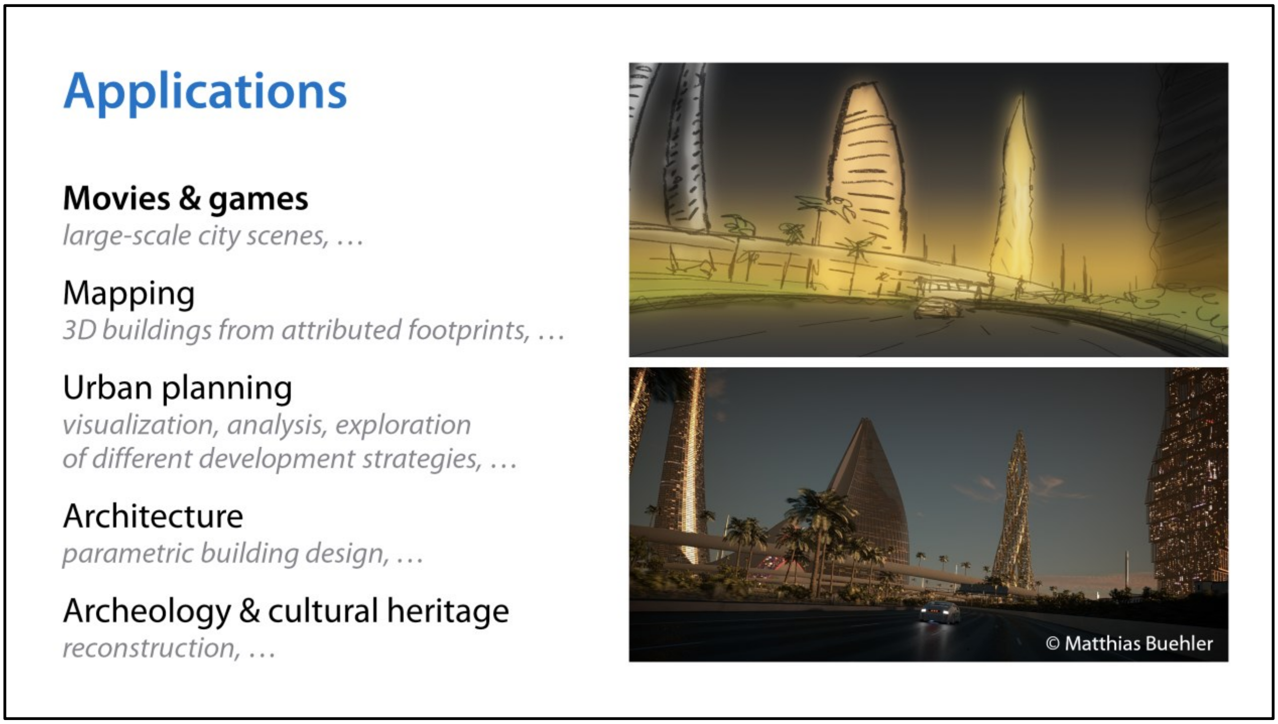
Example: rule-based modeling of facades

Rule: WndCol → split(y) { ~1 : Wall | 2 : Window | ~1 : Wall }



Properties and promises

Scalability	Large-scale generation of similar but varied objects • One potential answer to the ever-increasing demand for content
Compactness	Compressed representation • Example: building footprint + attributes + grammar
Descriptiveness	Describes the essence of a design ("recipe") • Can facilitate understanding and exploration
Flexibility	Adapt to different geometries and settings • Requires careful design
Reusability	"Model once, use many times"



Applications

Movies & games
large-scale city scenes, …

Mapping
3D buildings from attributed footprints, …

Urban planning
visualization, analysis, exploration
of different development strategies, …

Architecture
parametric building design, …

Archeology & cultural heritage
reconstruction, …

© Matthias Buehler

The images were kindly provided by Matthias Buehler (matthias.buehler@mac.com).

Applications

Movies & games
large-scale city scenes, …

Mapping
3D buildings from attributed footprints, …

Urban planning
visualization, analysis, exploration of different development strategies, …

Architecture
parametric building design, …

Archeology & cultural heritage
reconstruction, …

© Esri

Applications

Movies & games
large-scale city scenes, …

Mapping
3D buildings from attributed footprints, …

Urban planning
visualization, analysis, exploration of different development strategies, …

Architecture
parametric building design, …

Archeology & cultural heritage
reconstruction, …

Applications

Movies & games
large-scale city scenes, …

Mapping
3D buildings from attributed footprints, …

Urban planning
visualization, analysis, exploration
of different development strategies, …

Architecture
parametric building design, …

Archeology & cultural heritage
reconstruction, …

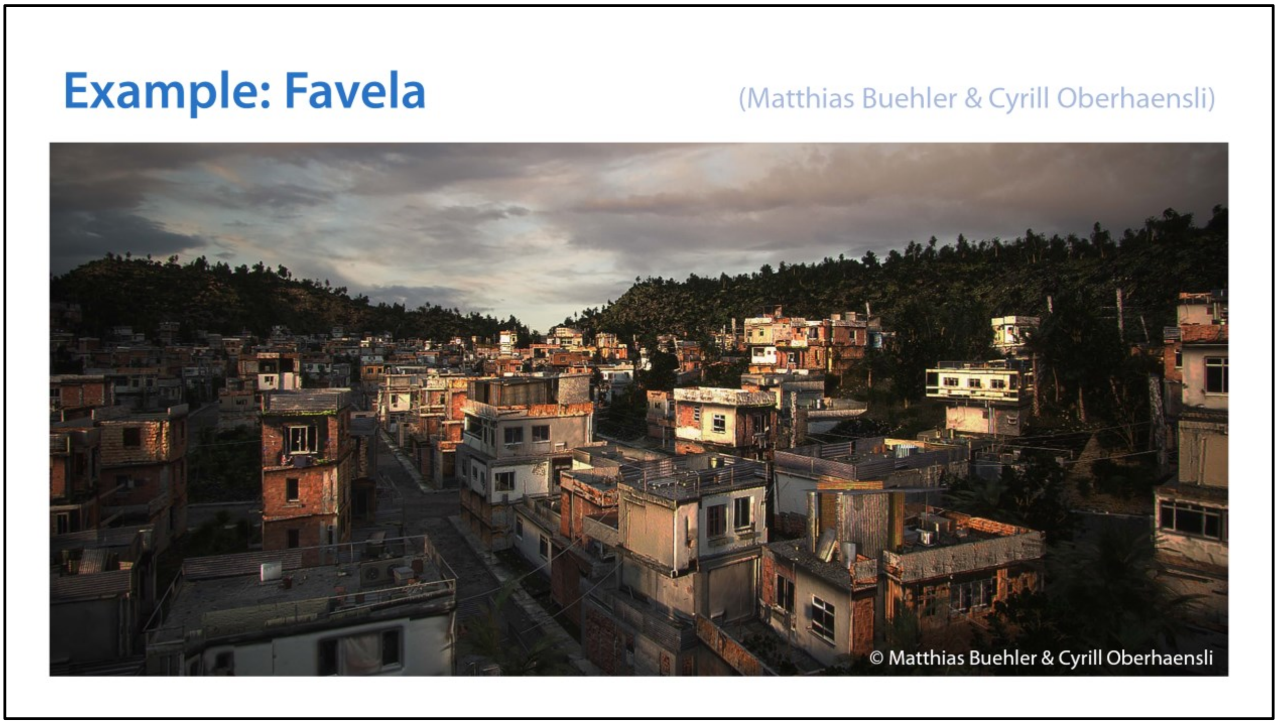
© Esri

Example: Favela

(Matthias Buehler & Cyrill Oberhaensli)

© Matthias Buehler & Cyrill Oberhaensli

A nice example that demonstrates what can be done with grammar-based procedural modeling techniques is the Favela project by Matthias Buehler (matthias.buehler@mac.com) and Cyrill Oberhaensli. Among others, it deals with hilly terrain and sloped building footprints, includes procedural vegetation, features cables and clotheslines, and involves the distribution of connection points and detail assets.



Example: Favela

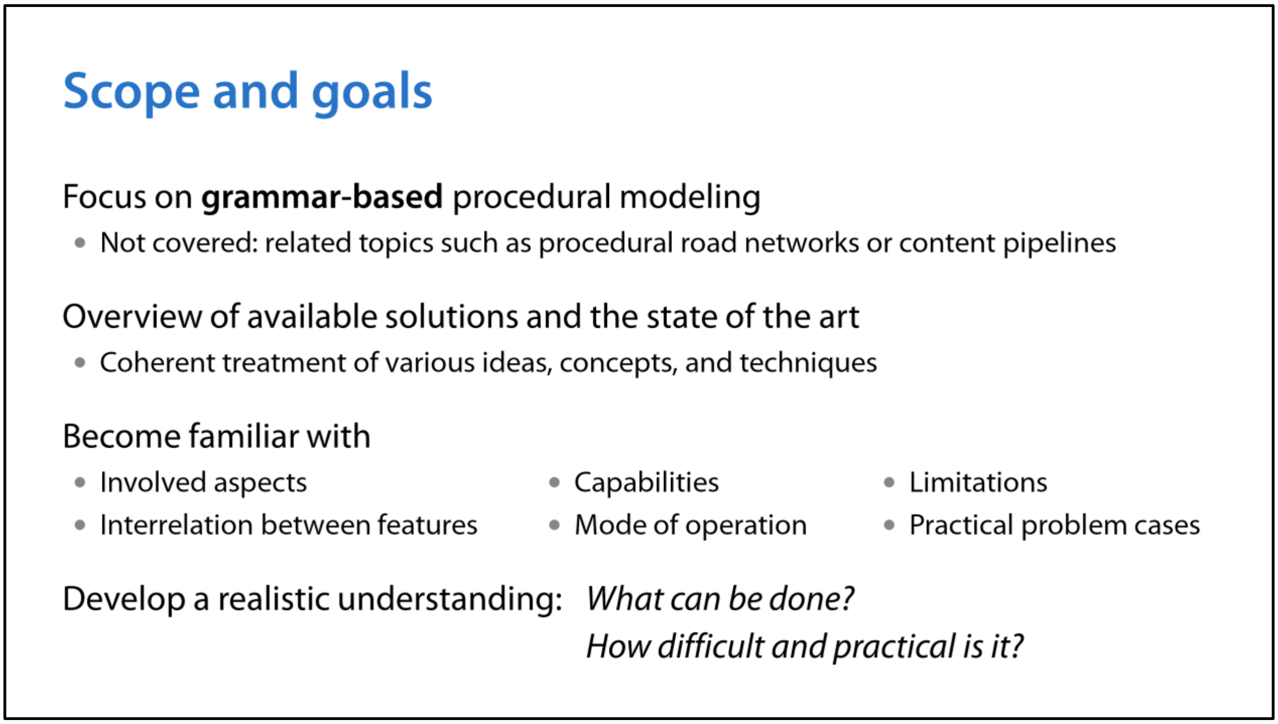
(Matthias Buehler & Cyrill Oberhaensli)

© Matthias Buehler & Cyrill Oberhaensli

Example: Favela

(Matthias Buehler & Cyrill Oberhaensli)

© Matthias Buehler & Cyrill Oberhaensli



Scope and goals

Focus on **grammar-based** procedural modeling

- Not covered: related topics such as procedural road networks or content pipelines

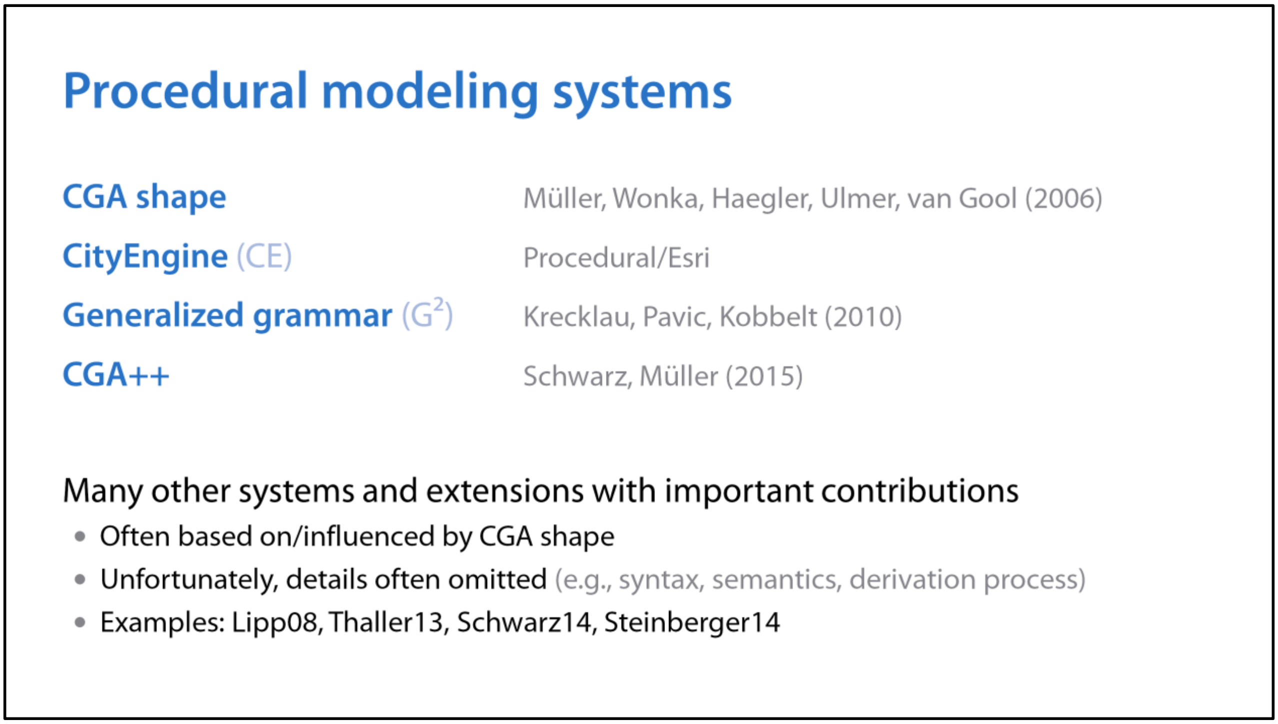
Overview of available solutions and the state of the art

- Coherent treatment of various ideas, concepts, and techniques

Become familiar with

- Involved aspects
- Interrelation between features
- Capabilities
- Mode of operation
- Limitations
- Practical problem cases

Develop a realistic understanding: *What can be done?*
How difficult and practical is it?



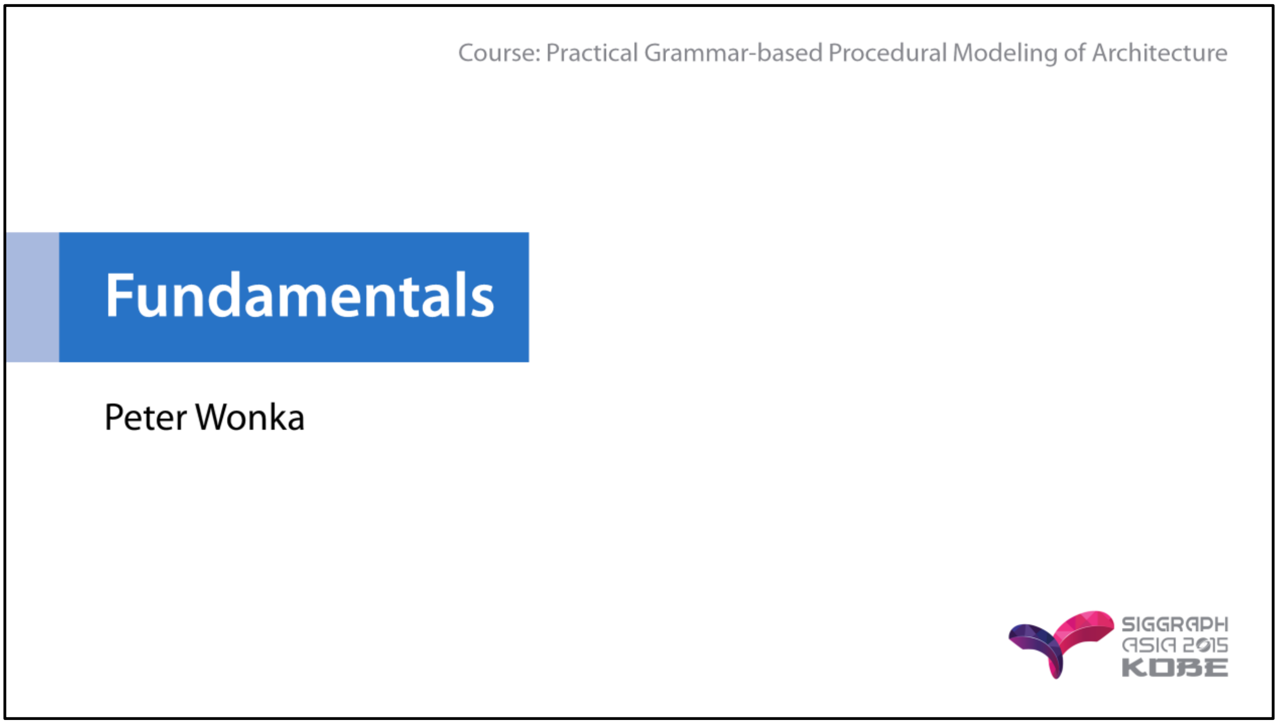
Procedural modeling systems

CGA shape	Müller, Wonka, Haegler, Ulmer, van Gool (2006)
CityEngine (CE)	Procedural/Esri
Generalized grammar (G^2)	Krecklau, Pavic, Kobbelt (2010)
CGA++	Schwarz, Müller (2015)

Many other systems and extensions with important contributions

- Often based on/influenced by CGA shape
- Unfortunately, details often omitted (e.g., syntax, semantics, derivation process)
- Examples: Lipp08, Thaller13, Schwarz14, Steinberger14

2 Fundamentals

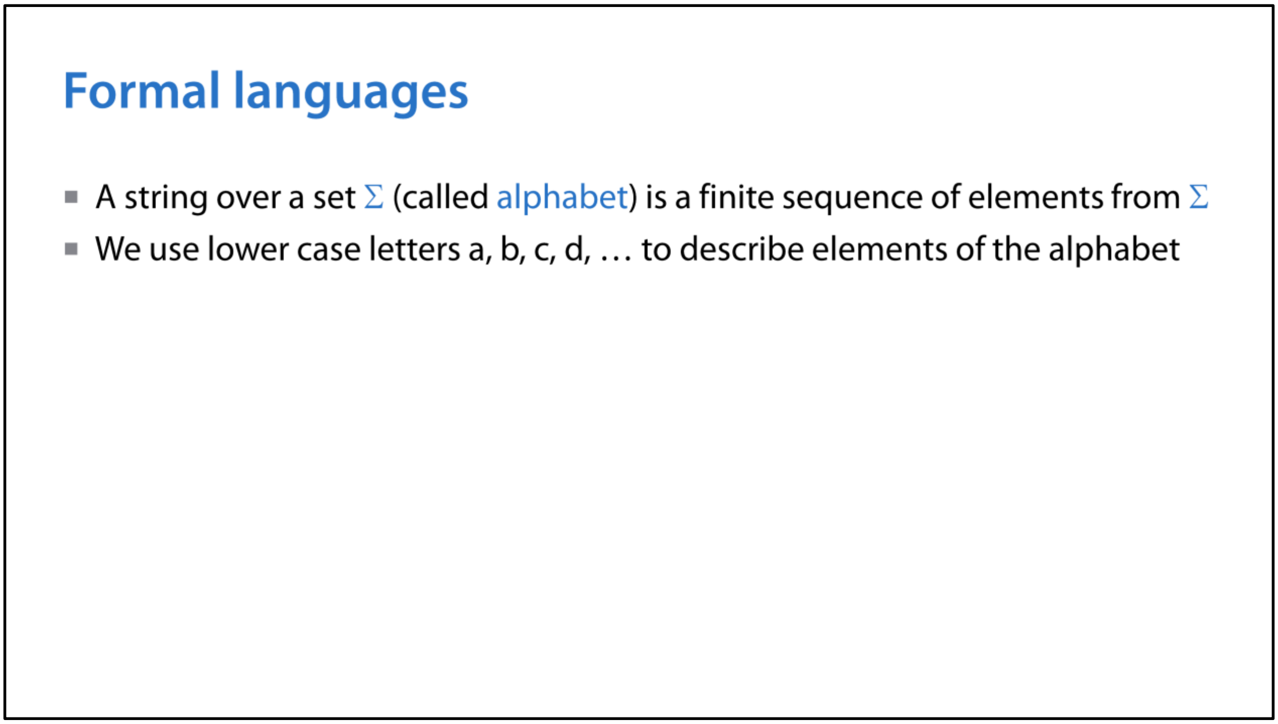


Course: Practical Grammar-based Procedural Modeling of Architecture

Fundamentals

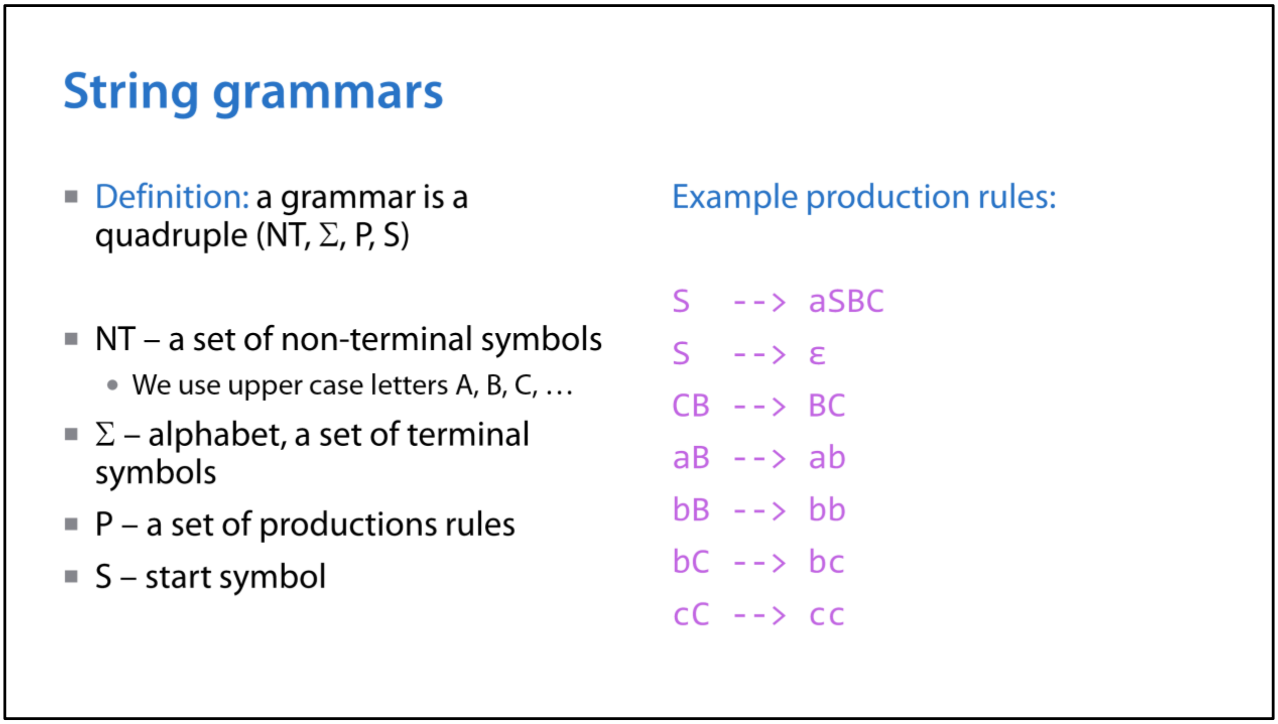
Peter Wonka

SIGGRAPH
ASIA 2015
KOBE



Formal languages

- A string over a set Σ (called alphabet) is a finite sequence of elements from Σ
- We use lower case letters a, b, c, d, … to describe elements of the alphabet

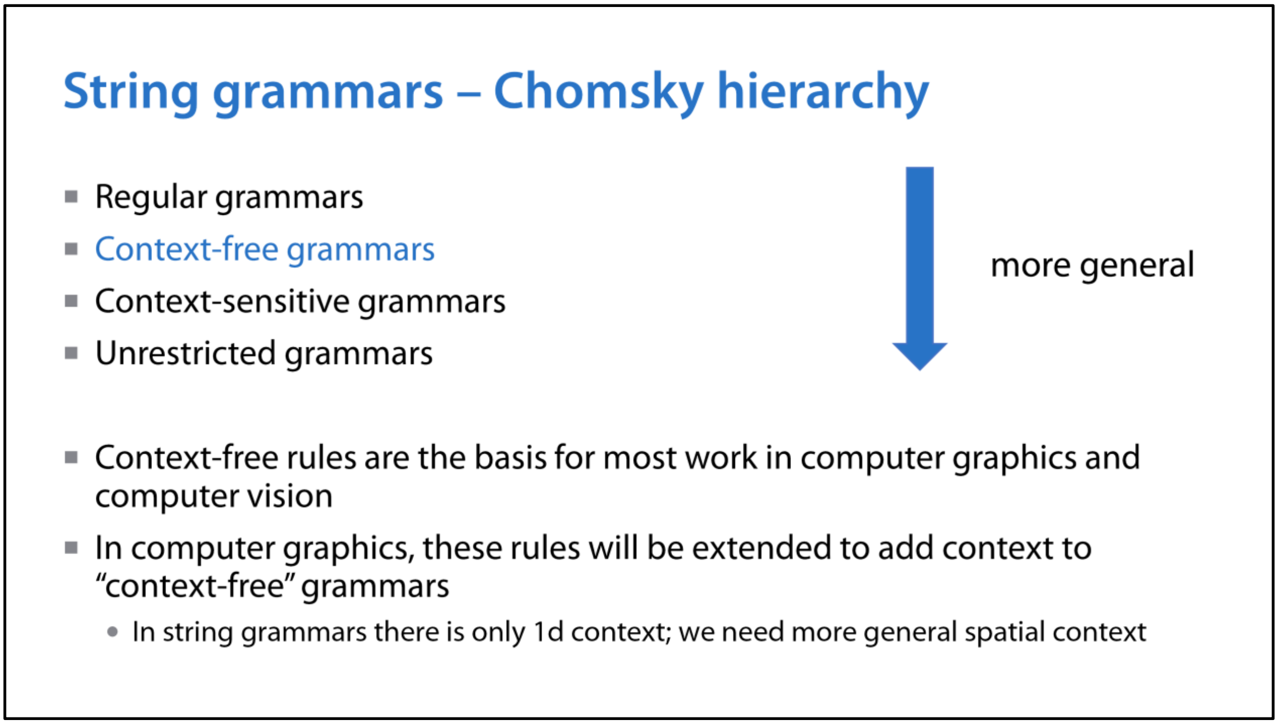


String grammars

- Definition: a grammar is a quadruple (NT, Σ, P, S)
- NT – a set of non-terminal symbols
  - We use upper case letters A, B, C, …
- Σ – alphabet, a set of terminal symbols
- P – a set of productions rules
- S – start symbol

Example production rules:

```
S   --> aSBC
S   --> ε
CB  --> BC
aB  --> ab
bB  --> bb
bC  --> bc
cC  --> cc
```

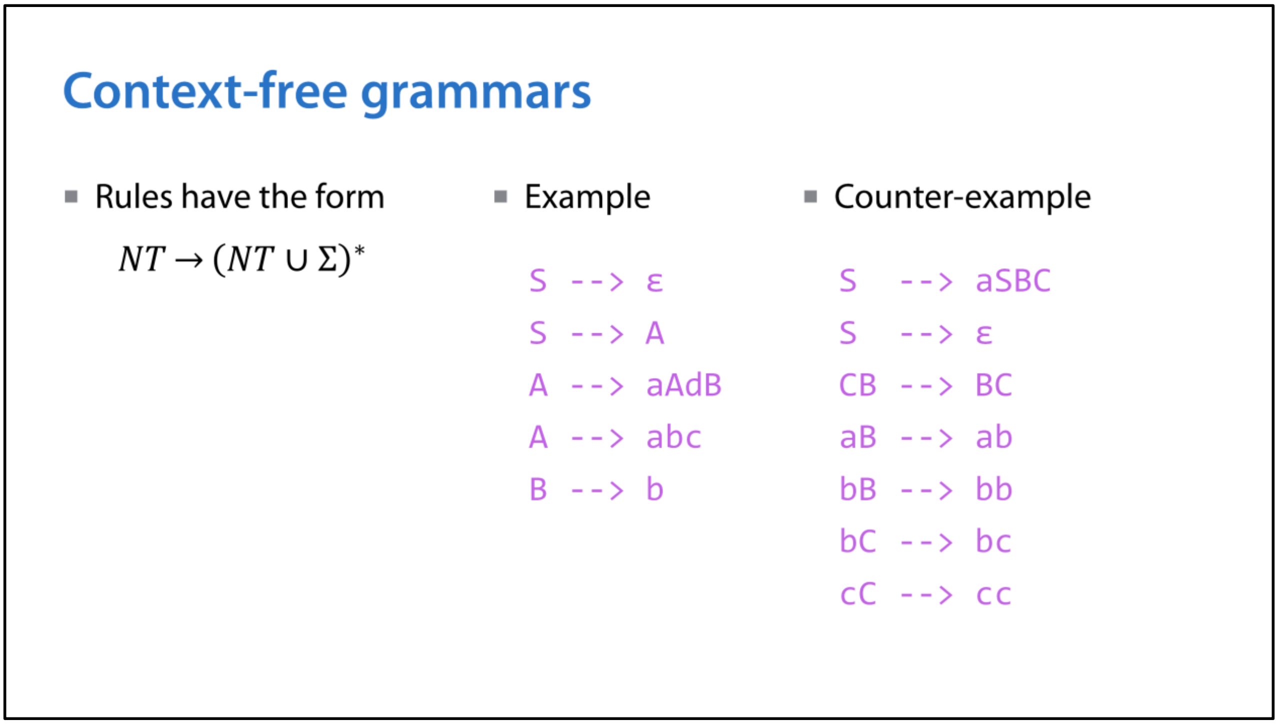


String grammars – Chomsky hierarchy

- Regular grammars
- Context-free grammars
- Context-sensitive grammars
- Unrestricted grammars

more general

- Context-free rules are the basis for most work in computer graphics and computer vision
- In computer graphics, these rules will be extended to add context to “context-free” grammars
  - In string grammars there is only 1d context; we need more general spatial context



Context-free grammars

- Rules have the form

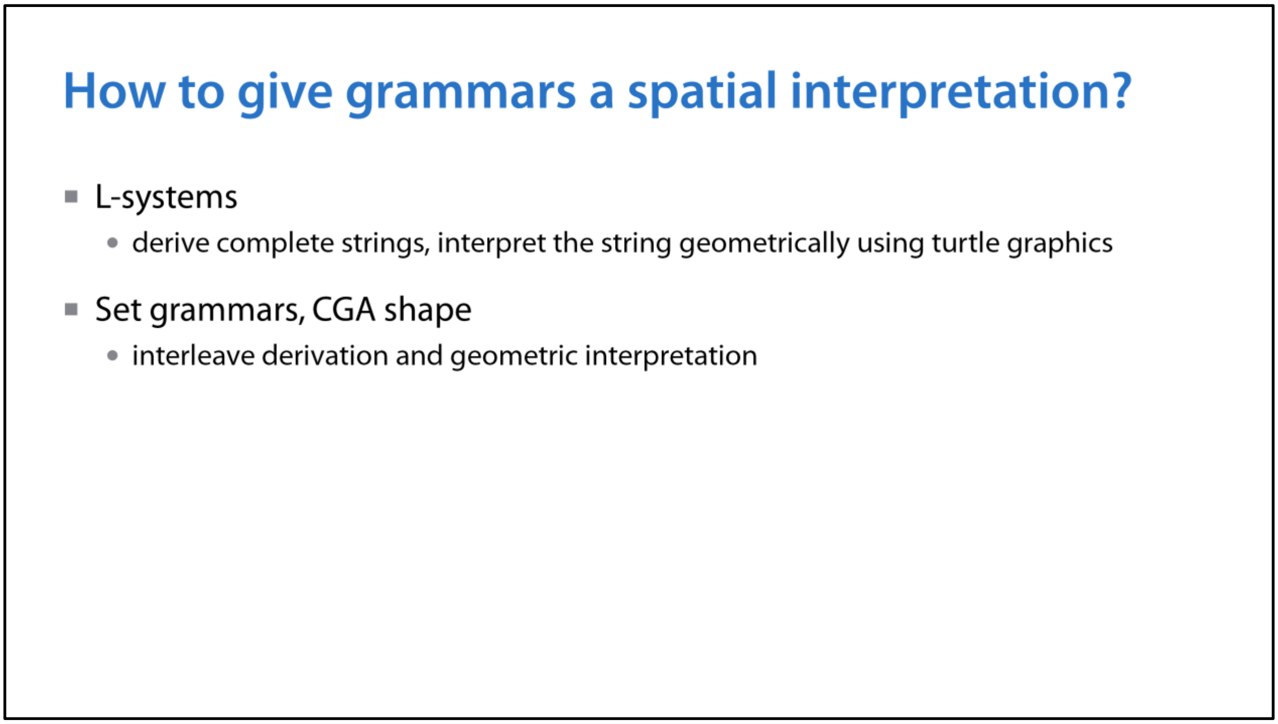
$$NT \rightarrow (NT \cup \Sigma)^*$$

- Example

```
S --> ε
S --> A
A --> aAdB
A --> abc
B --> b
```

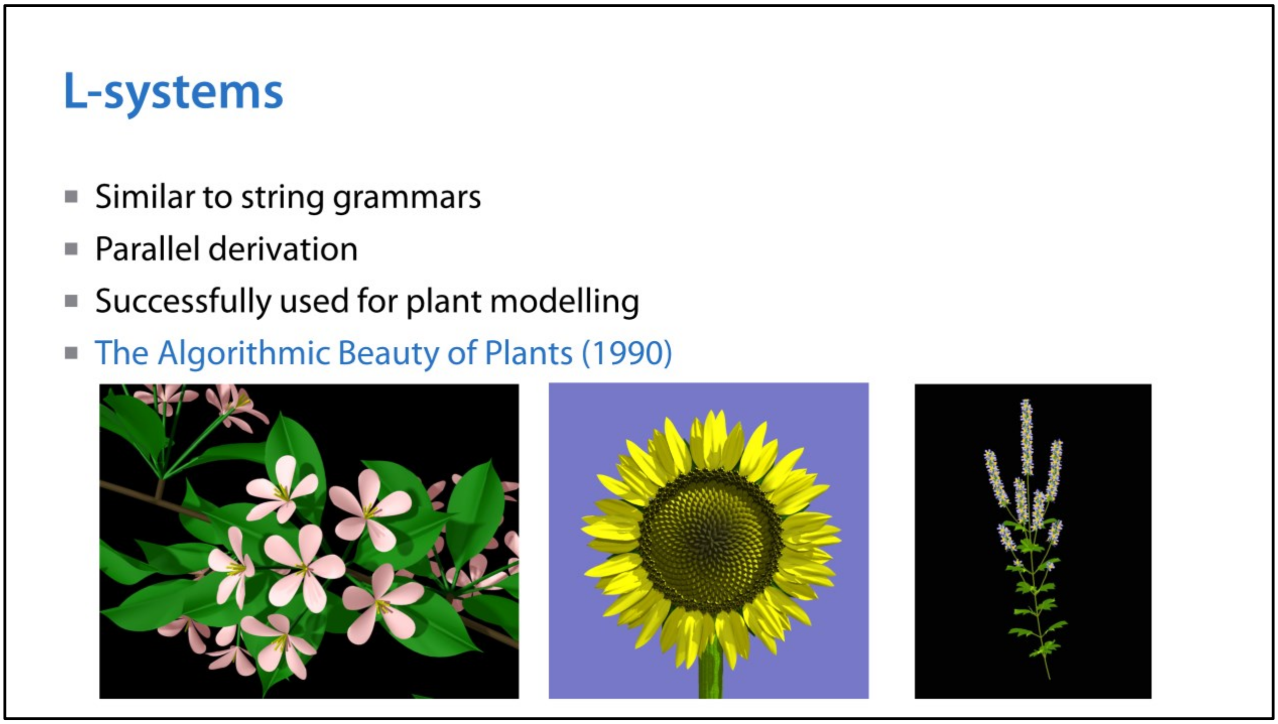
- Counter-example

```
S  --> aSBC
S  --> ε
CB --> BC
aB --> ab
bB --> bb
bC --> bc
cC --> cc
```



How to give grammars a spatial interpretation?

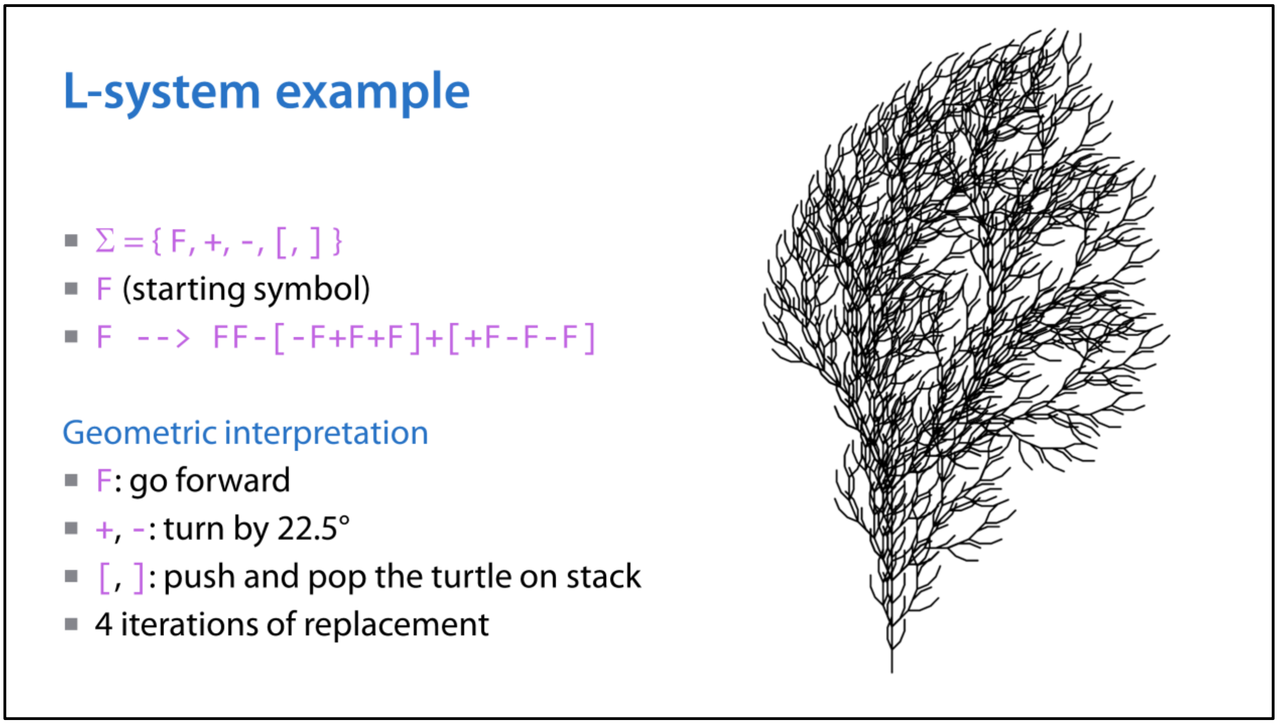
- L-systems
  - derive complete strings, interpret the string geometrically using turtle graphics
- Set grammars, CGA shape
  - interleave derivation and geometric interpretation



L-systems

- Similar to string grammars
- Parallel derivation
- Successfully used for plant modelling
- The Algorithmic Beauty of Plants (1990)

Images: The Algorithmic Beauty of Plants 1990



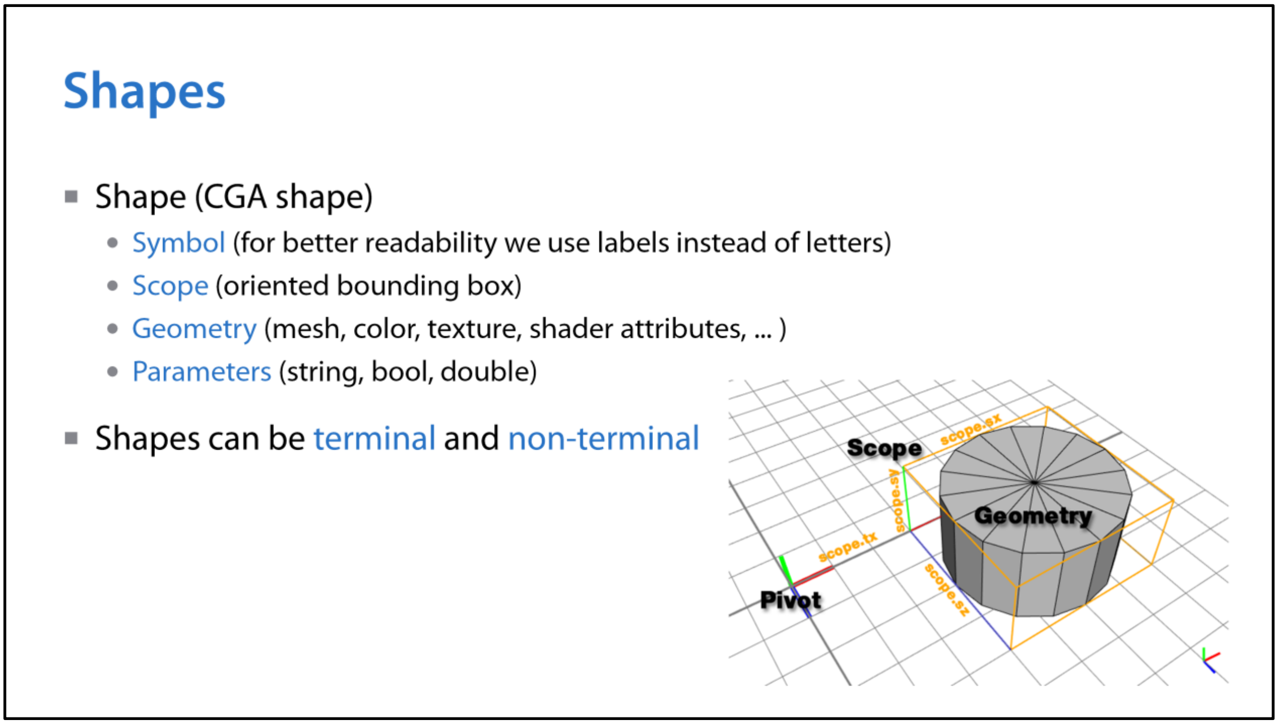
L-system example

- $\Sigma = \{ F, +, -, [,] \}$
- F (starting symbol)
- `F --> FF-[-F+F+F]+[+F-F-F]`

Geometric interpretation

- F: go forward
- +, -: turn by 22.5°
- [,]: push and pop the turtle on stack
- 4 iterations of replacement

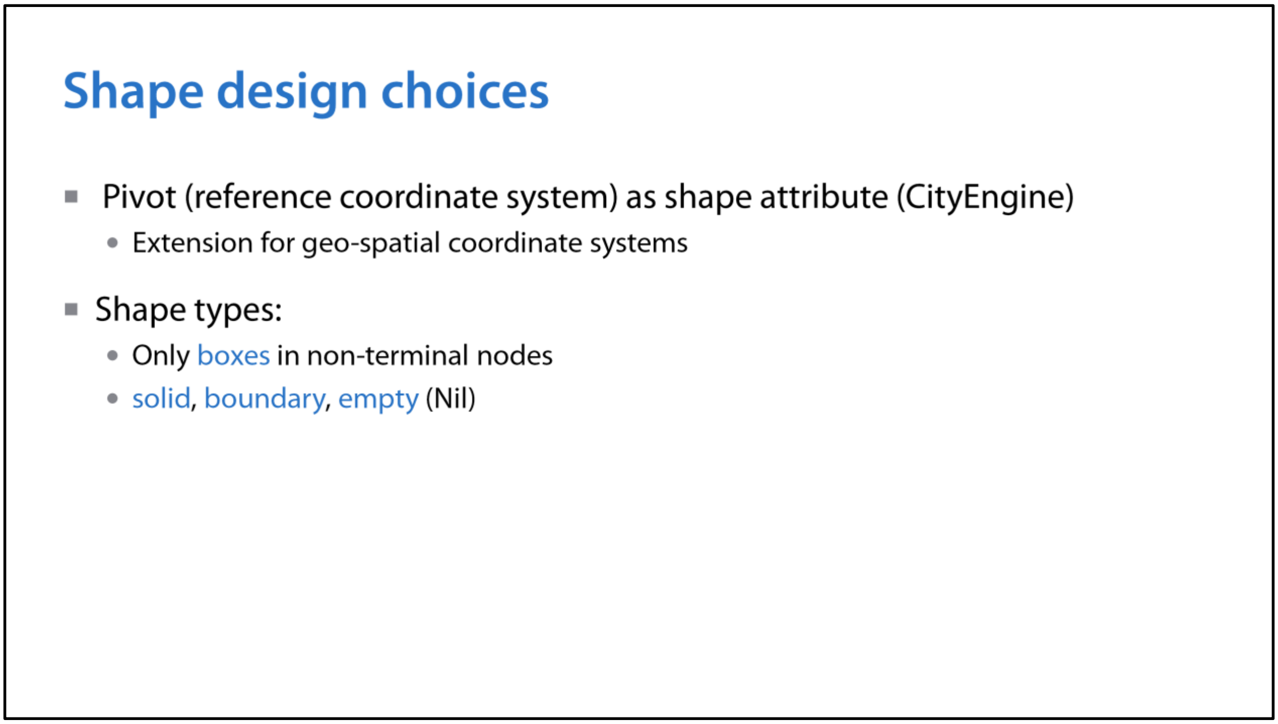
Image: The Algorithmic Beauty of Plants 1990



Shapes

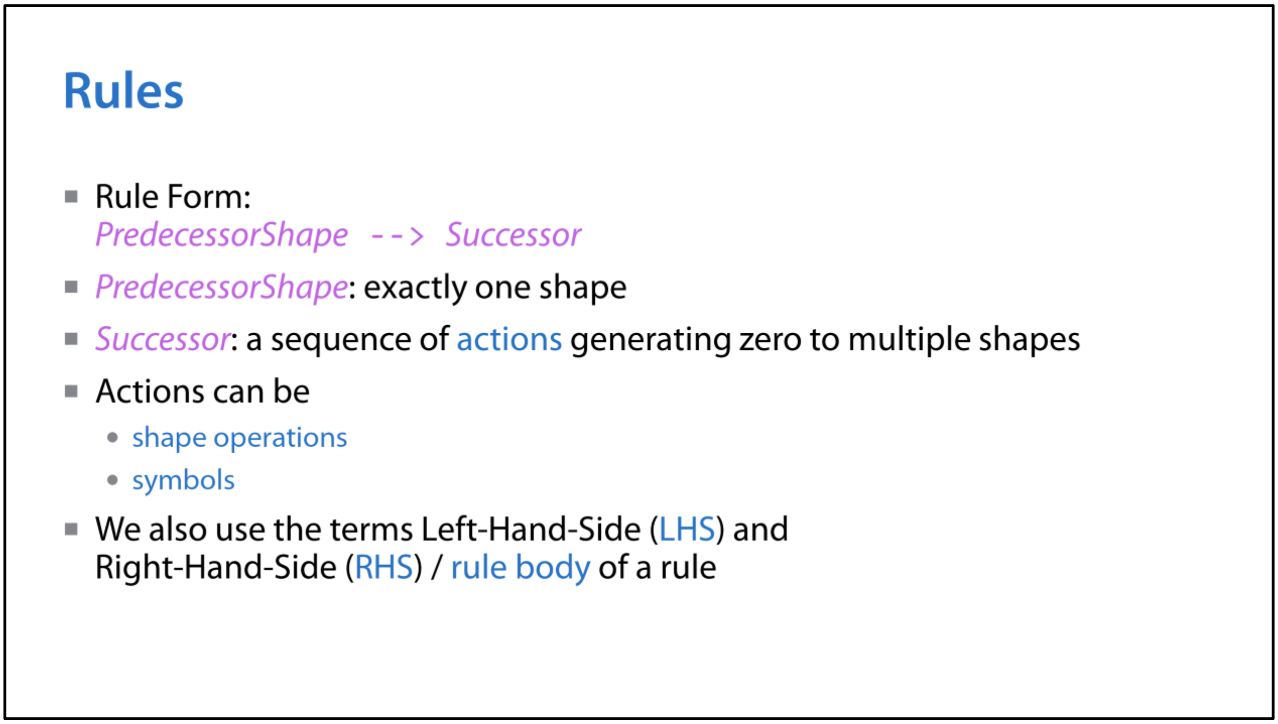
- Shape (CGA shape)
  - Symbol (for better readability we use labels instead of letters)
  - Scope (oriented bounding box)
  - Geometry (mesh, color, texture, shader attributes, ...)
  - Parameters (string, bool, double)
- Shapes can be terminal and non-terminal

Image: CityEngine Online Help System



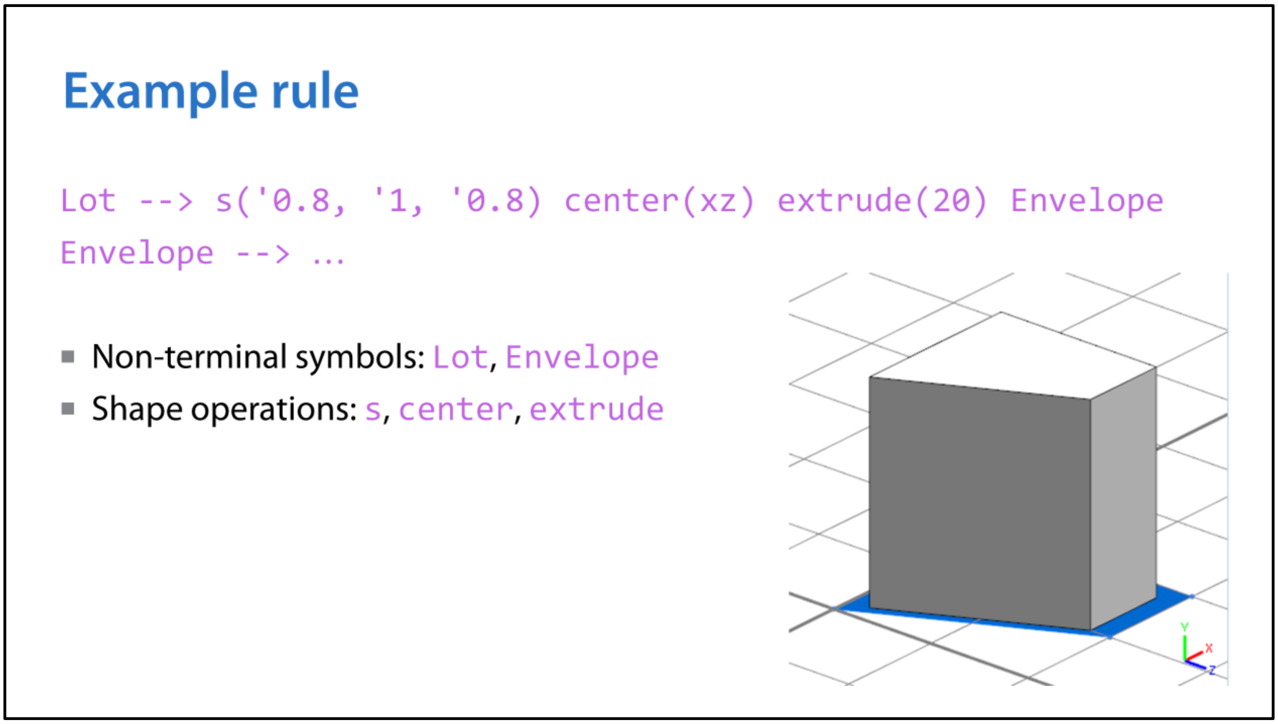
Shape design choices

- Pivot (reference coordinate system) as shape attribute (CityEngine)
  - Extension for geo-spatial coordinate systems
- Shape types:
  - Only boxes in non-terminal nodes
  - solid, boundary, empty (Nil)



Rules

- Rule Form:
  PredecessorShape --> Successor
- *PredecessorShape*: exactly one shape
- *Successor*: a sequence of actions generating zero to multiple shapes
- Actions can be
  - shape operations
  - symbols
- We also use the terms Left-Hand-Side (LHS) and Right-Hand-Side (RHS) / rule body of a rule

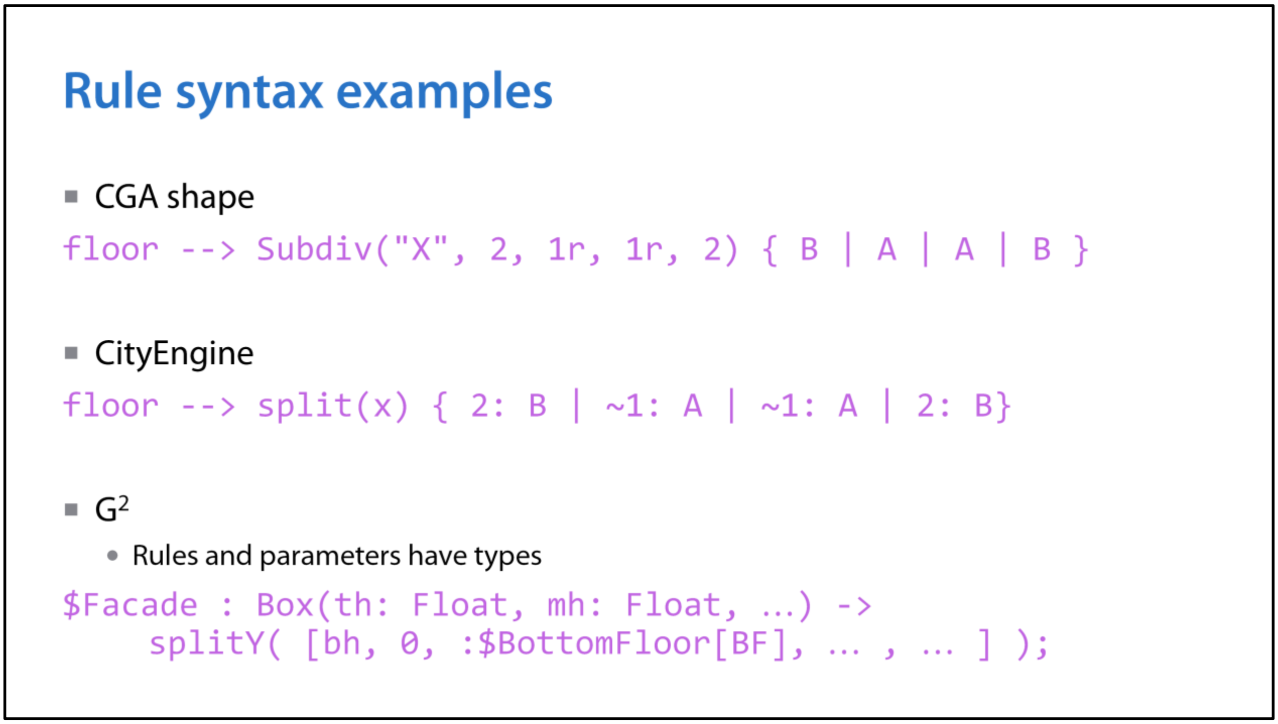


Example rule

```
Lot --> s('0.8, '1, '0.8) center(xz) extrude(20) Envelope
Envelope --> …
```

- Non-terminal symbols: `Lot`, `Envelope`
- Shape operations: `s`, `center`, `extrude`

Image: CityEngine Online Help System



Rule syntax examples

- CGA shape

```
floor --> Subdiv("X", 2, 1r, 1r, 2) { B | A | A | B }
```

- CityEngine

```
floor --> split(x) { 2: B | ~1: A | ~1: A | 2: B}
```

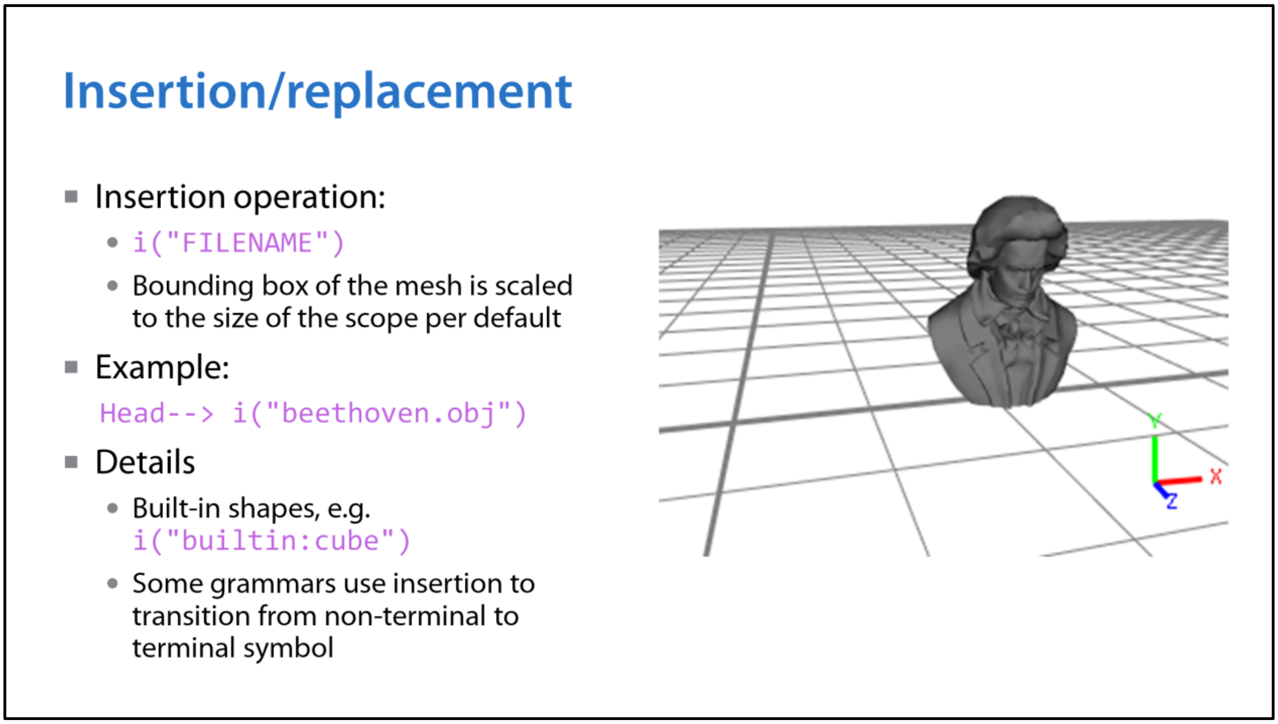
- G^2
  - Rules and parameters have types

```
$Facade : Box(th: Float, mh: Float, …) ->
    splitY( [bh, 0, :$BottomFloor[BF], … , … ] );
```

Elementary shape operations

- Insertion of assets
- Transformations
- Extrusion
- Center
- Component split
- Subdivision split

- Most examples use CityEngine syntax



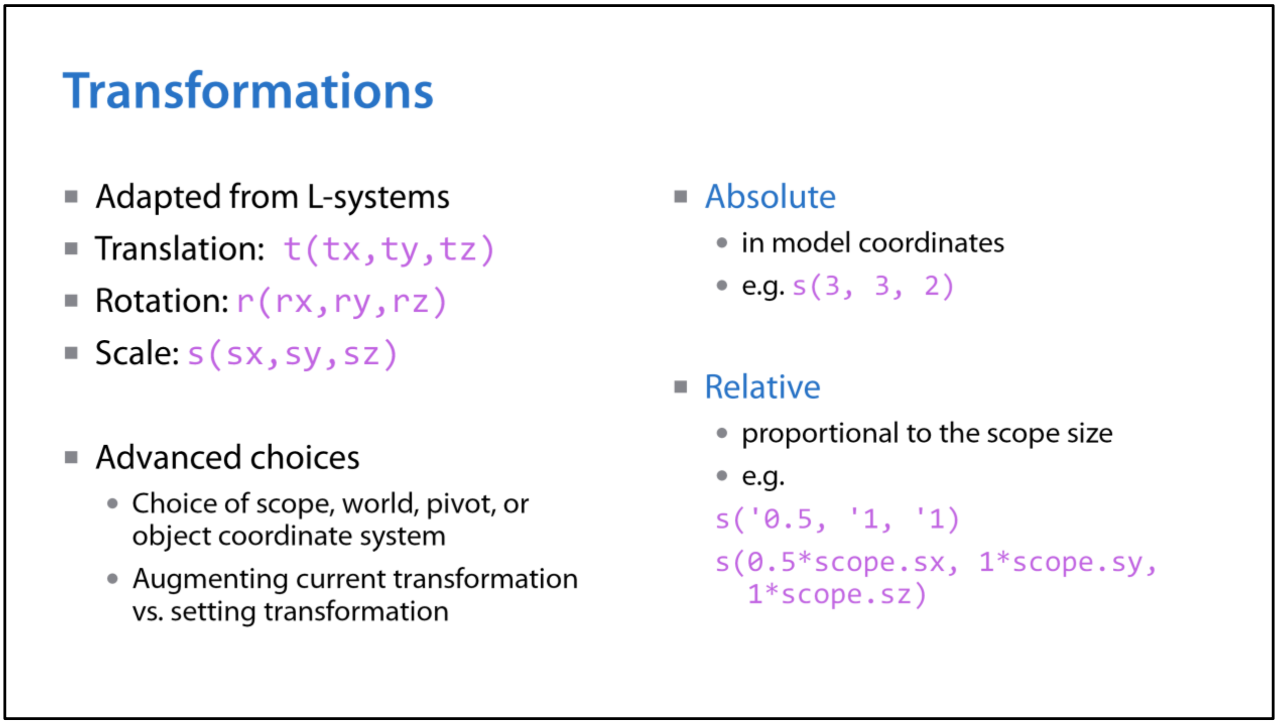
Insertion/replacement

- Insertion operation:
  - `i("FILENAME")`
  - Bounding box of the mesh is scaled to the size of the scope per default
- Example:

```
Head--> i("beethoven.obj")
```

- Details
  - Built-in shapes, e.g. `i("builtin:cube")`
  - Some grammars use insertion to transition from non-terminal to terminal symbol

Image: CityEngine Online Help System



Transformations

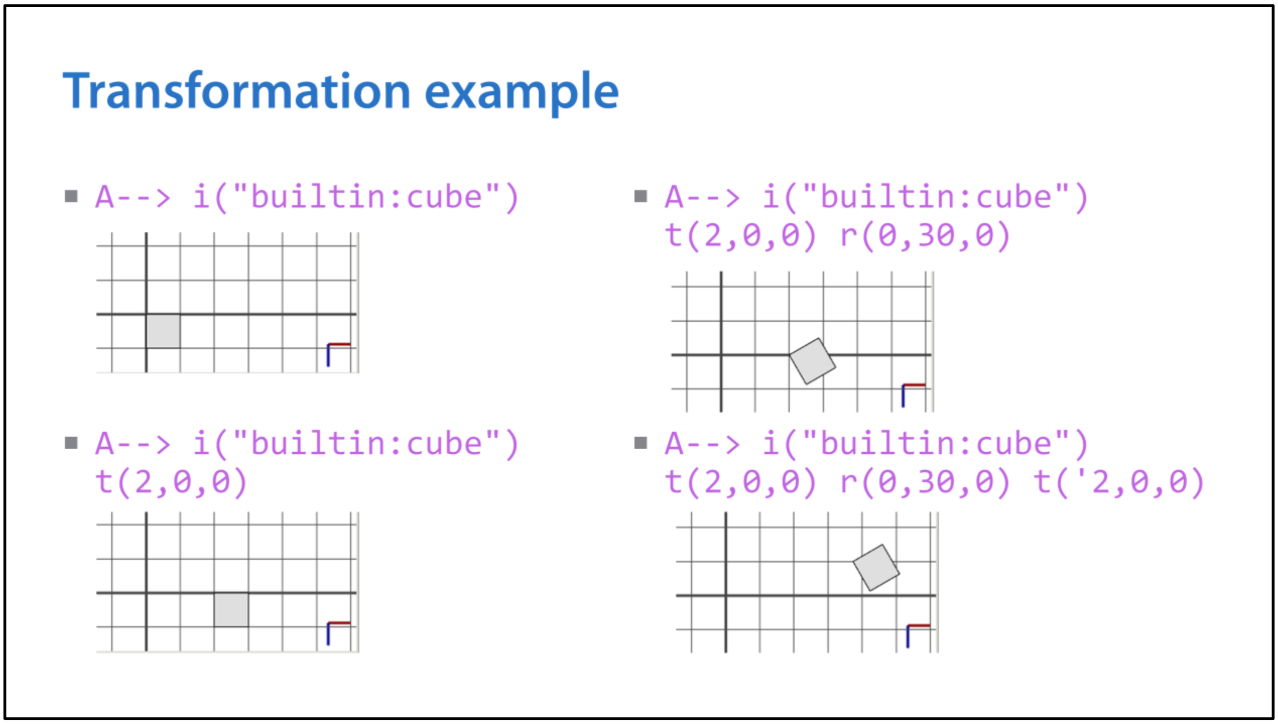
- Adapted from L-systems
- Translation: `t(tx,ty,tz)`
- Rotation: `r(rx,ry,rz)`
- Scale: `s(sx,sy,sz)`

- Advanced choices
  - Choice of scope, world, pivot, or object coordinate system
  - Augmenting current transformation vs. setting transformation

- Absolute
  - in model coordinates
  - e.g. `s(3, 3, 2)`

- Relative
  - proportional to the scope size
  - e.g.

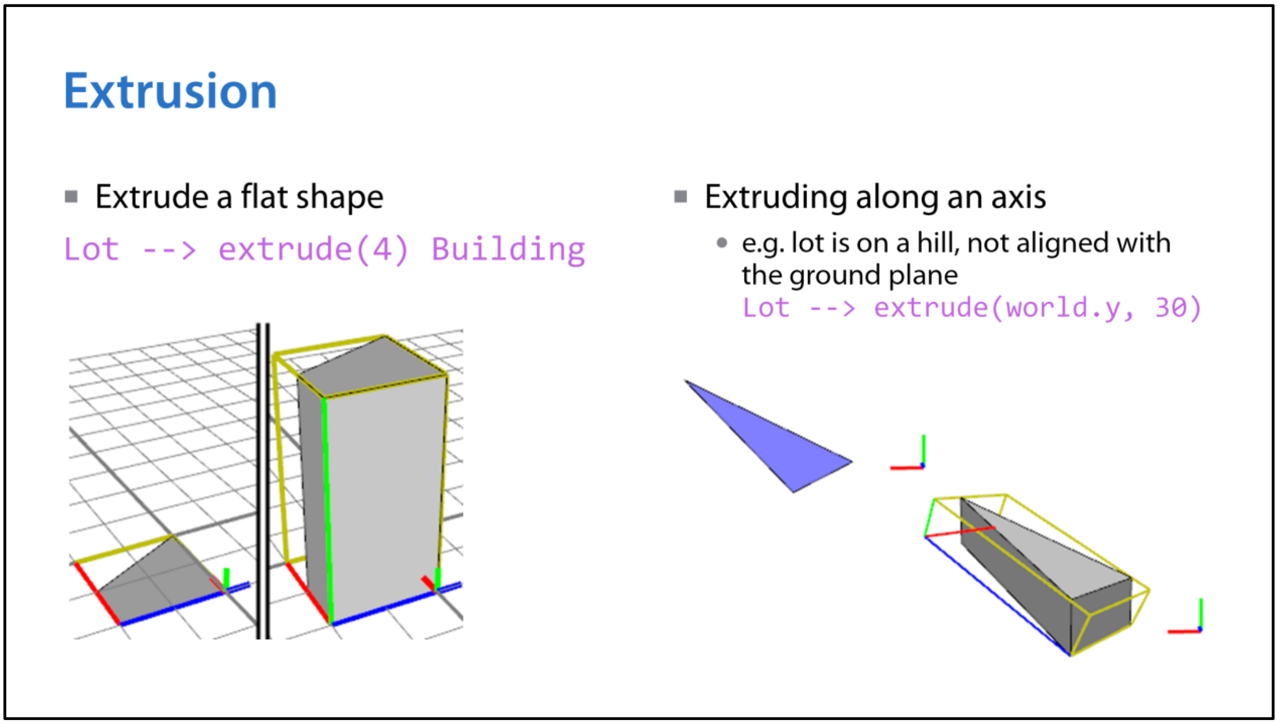
```
s('0.5, '1, '1)
s(0.5*scope.sx, 1*scope.sy,
  1*scope.sz)
```



Transformation example

- `A--> i("builtin:cube")`
- `A--> i("builtin:cube") t(2,0,0)`
- `A--> i("builtin:cube") t(2,0,0) r(0,30,0)`
- `A--> i("builtin:cube") t(2,0,0) r(0,30,0) t('2,0,0)`

Image: CityEngine Online Help System



Extrusion

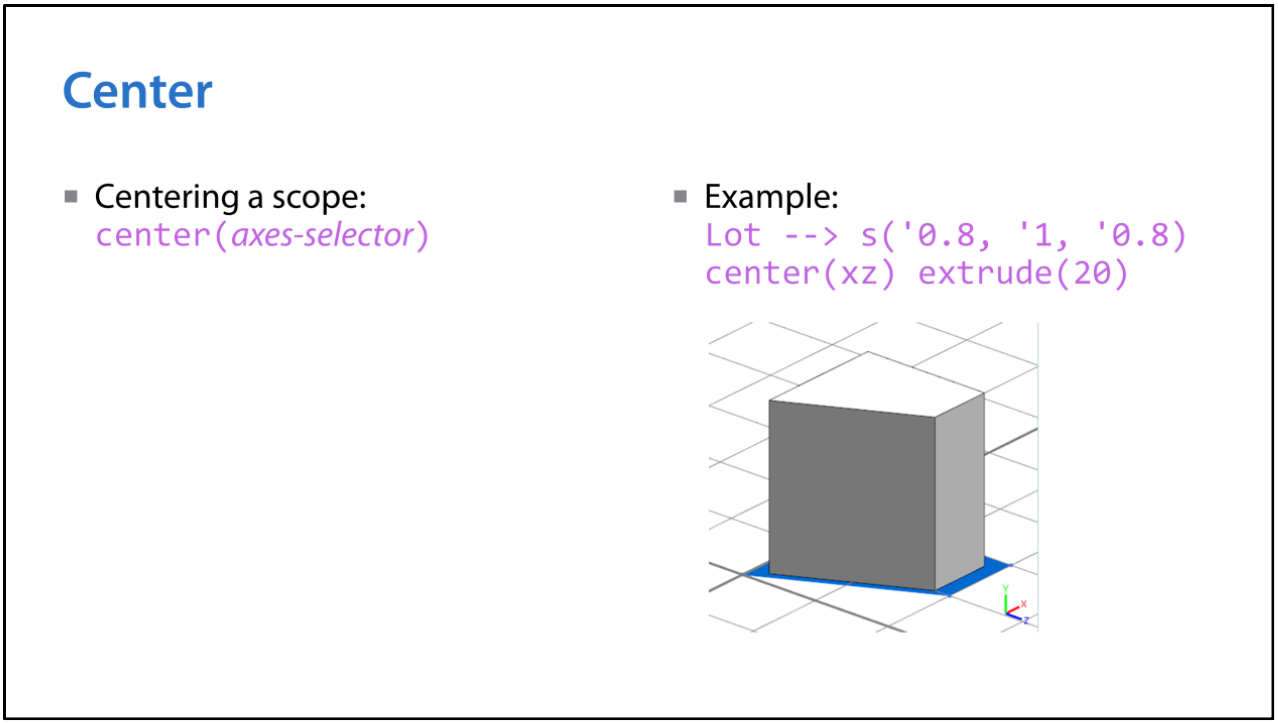
- Extrude a flat shape

`Lot --> extrude(4) Building`

- Extruding along an axis
  - e.g. lot is on a hill, not aligned with the ground plane

`Lot --> extrude(world.y, 30)`

Image: CityEngine Online Help System



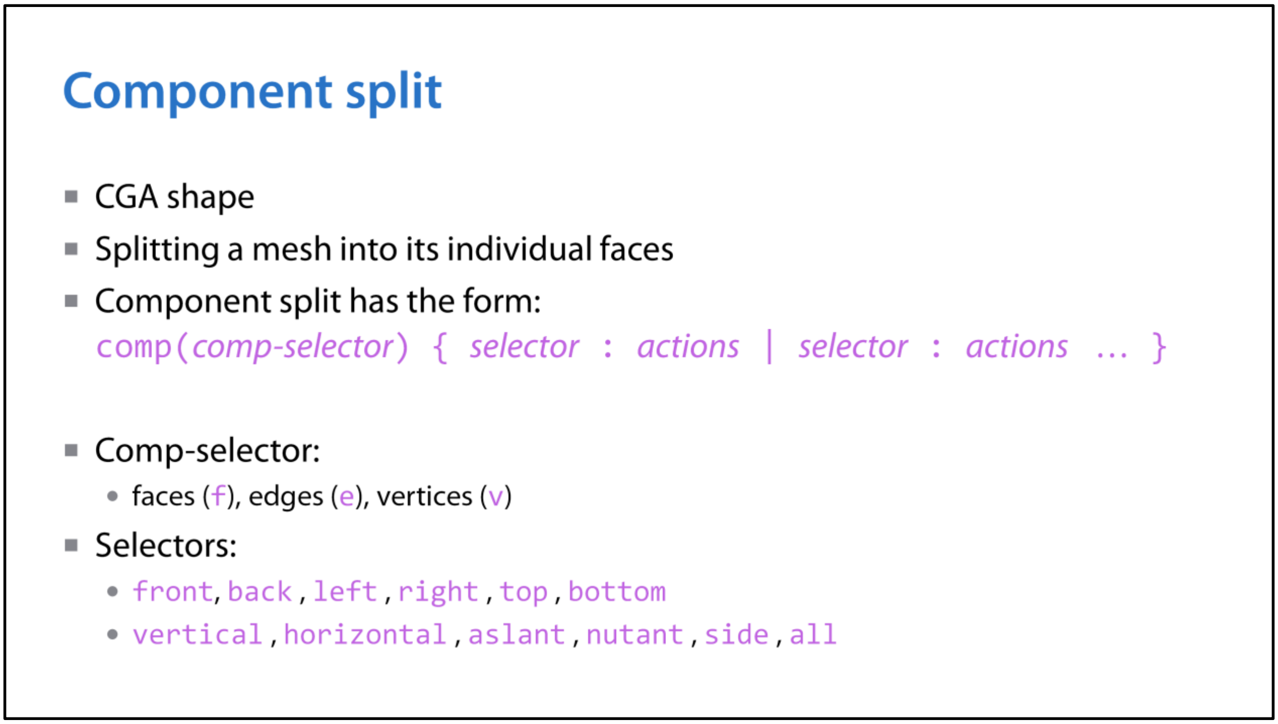
Center

- Centering a scope:
  `center(`*`axes-selector`*`)`

- Example:

```
Lot --> s('0.8, '1, '0.8)
center(xz) extrude(20)
```

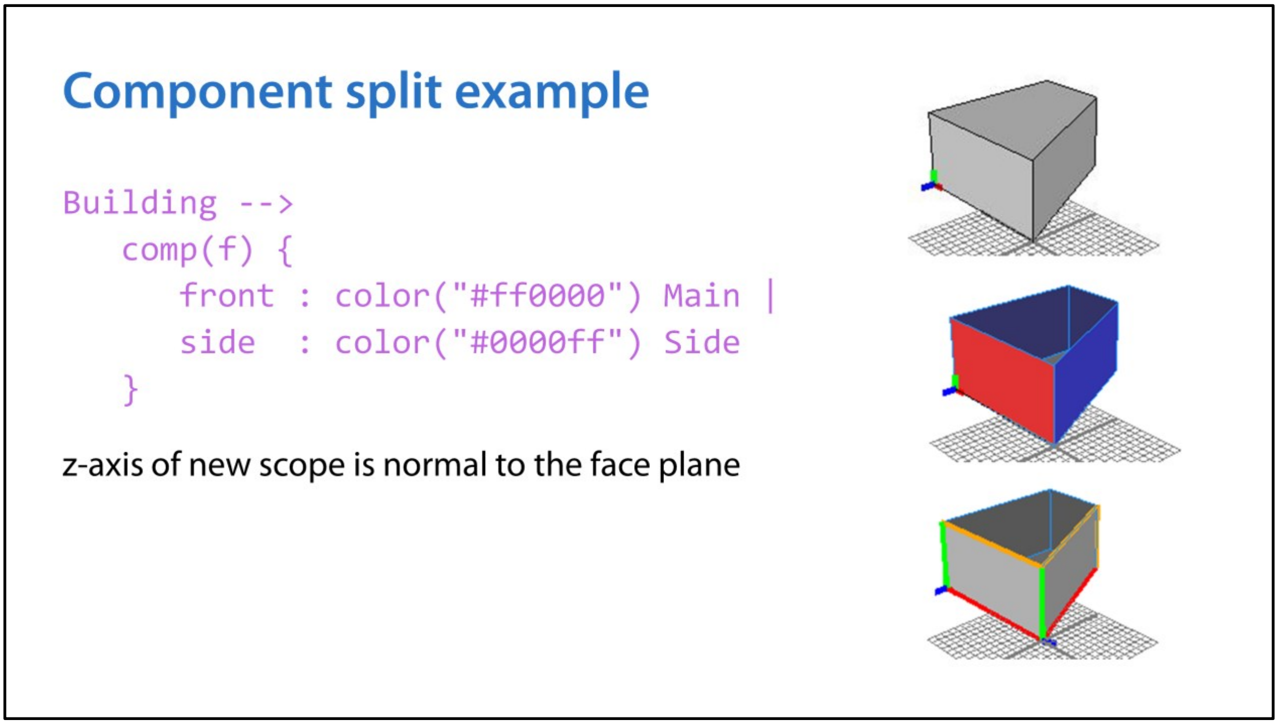
Image: CityEngine Online Help System



Component split

- CGA shape
- Splitting a mesh into its individual faces
- Component split has the form:
  `comp(comp-selector) { selector : actions | selector : actions … }`

- Comp-selector:
  - faces (`f`), edges (`e`), vertices (`v`)
- Selectors:
  - `front`, `back`, `left`, `right`, `top`, `bottom`
  - `vertical`, `horizontal`, `aslant`, `nutant`, `side`, `all`

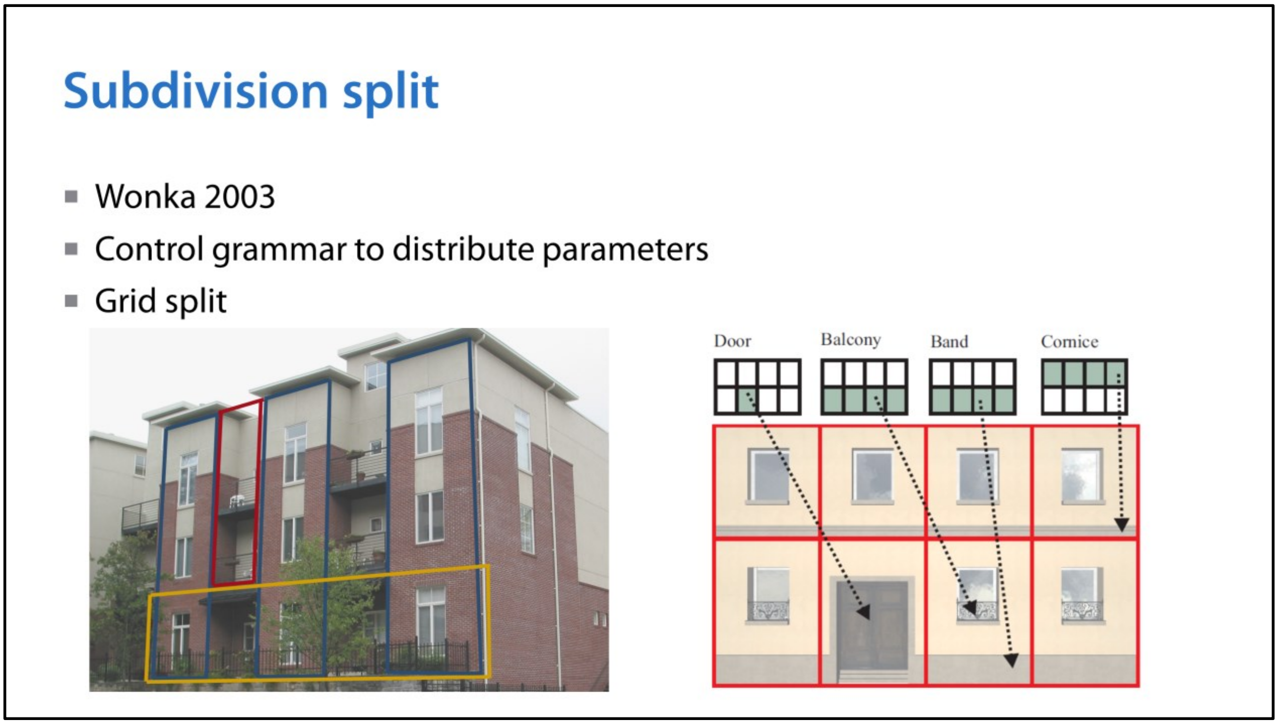


Component split example

```
Building -->
   comp(f) {
      front : color("#ff0000") Main |
      side  : color("#0000ff") Side
   }
```

z-axis of new scope is normal to the face plane

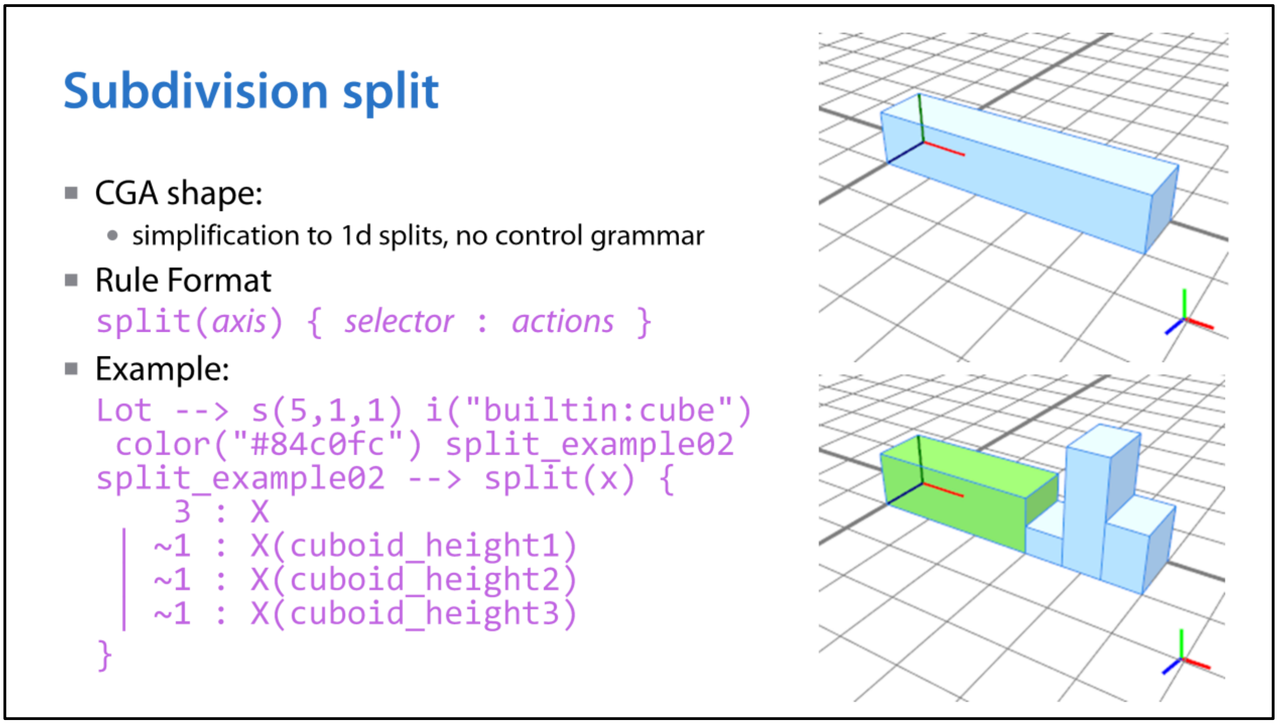
Image: CityEngine Online Help System



Subdivision split

- Wonka 2003
- Control grammar to distribute parameters
- Grid split

Image: CityEngine Online Help System



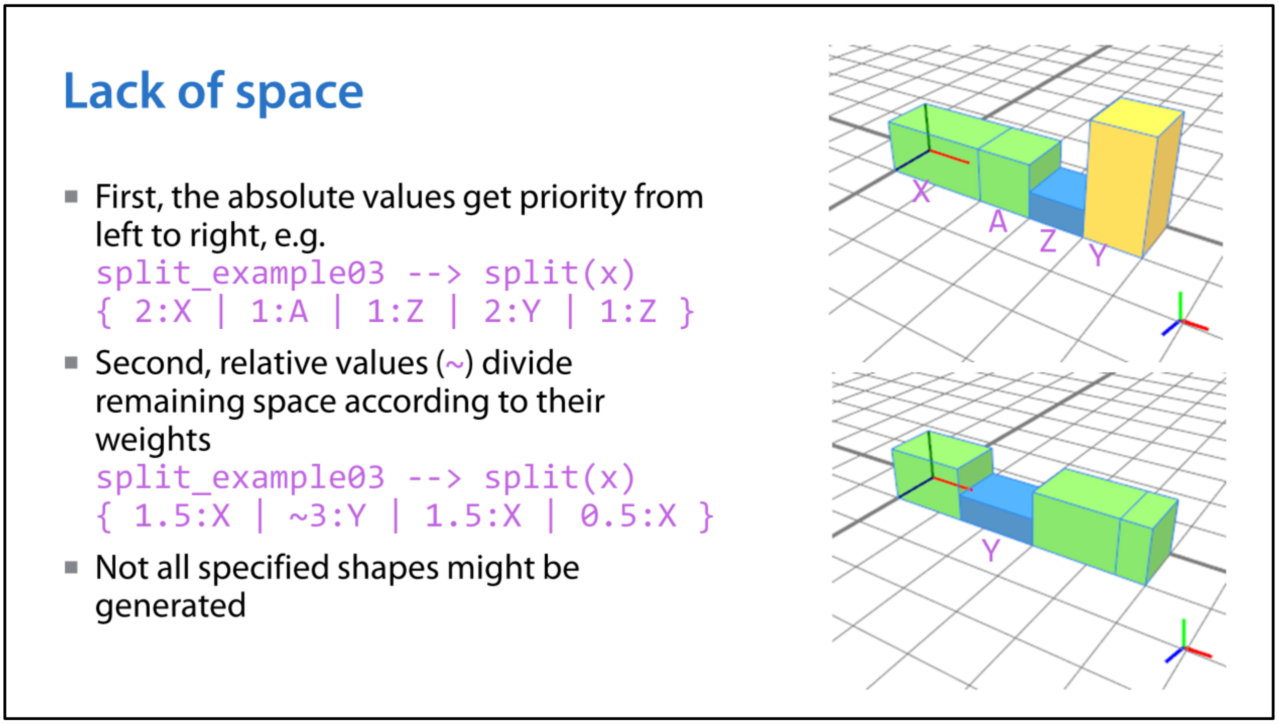
Subdivision split

- CGA shape:
  - simplification to 1d splits, no control grammar
- Rule Format

  `split(axis) { selector : actions }`
- Example:

```
Lot --> s(5,1,1) i("builtin:cube")
 color("#84c0fc") split_example02
split_example02 --> split(x) {
    3 : X
  | ~1 : X(cuboid_height1)
  | ~1 : X(cuboid_height2)
  | ~1 : X(cuboid_height3)
}
```

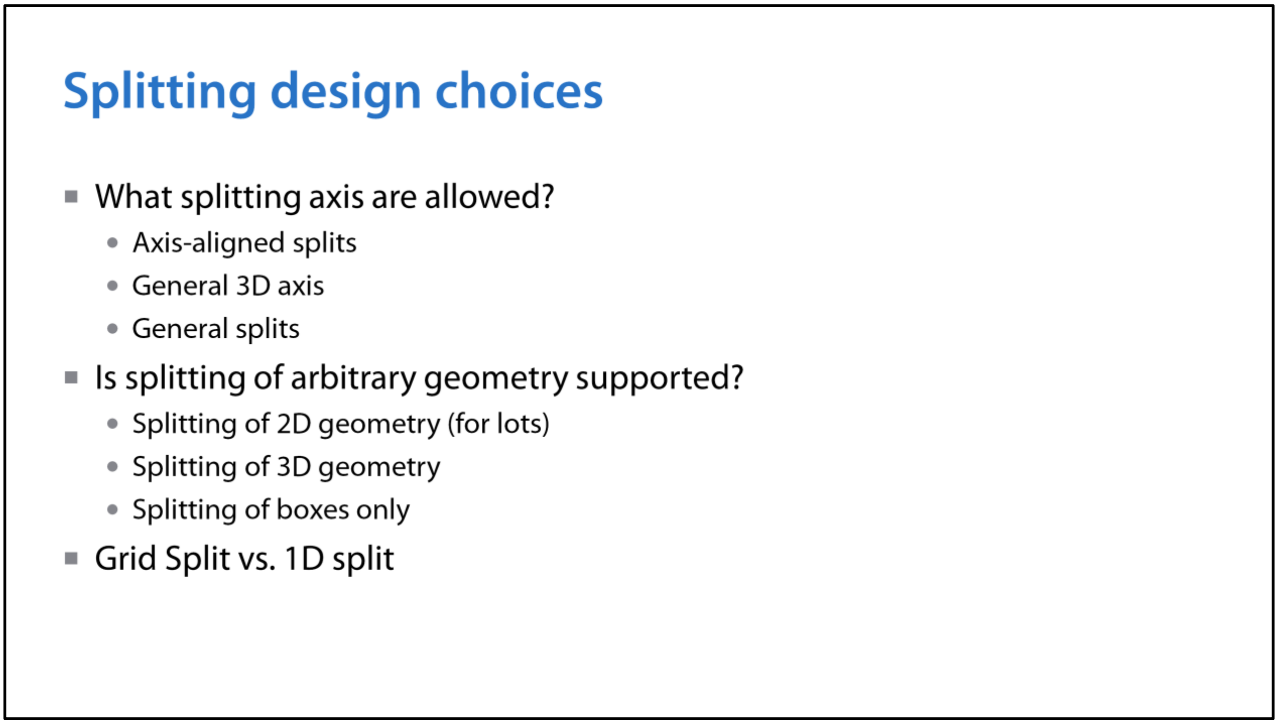
Image: CityEngine Online Help System



Lack of space

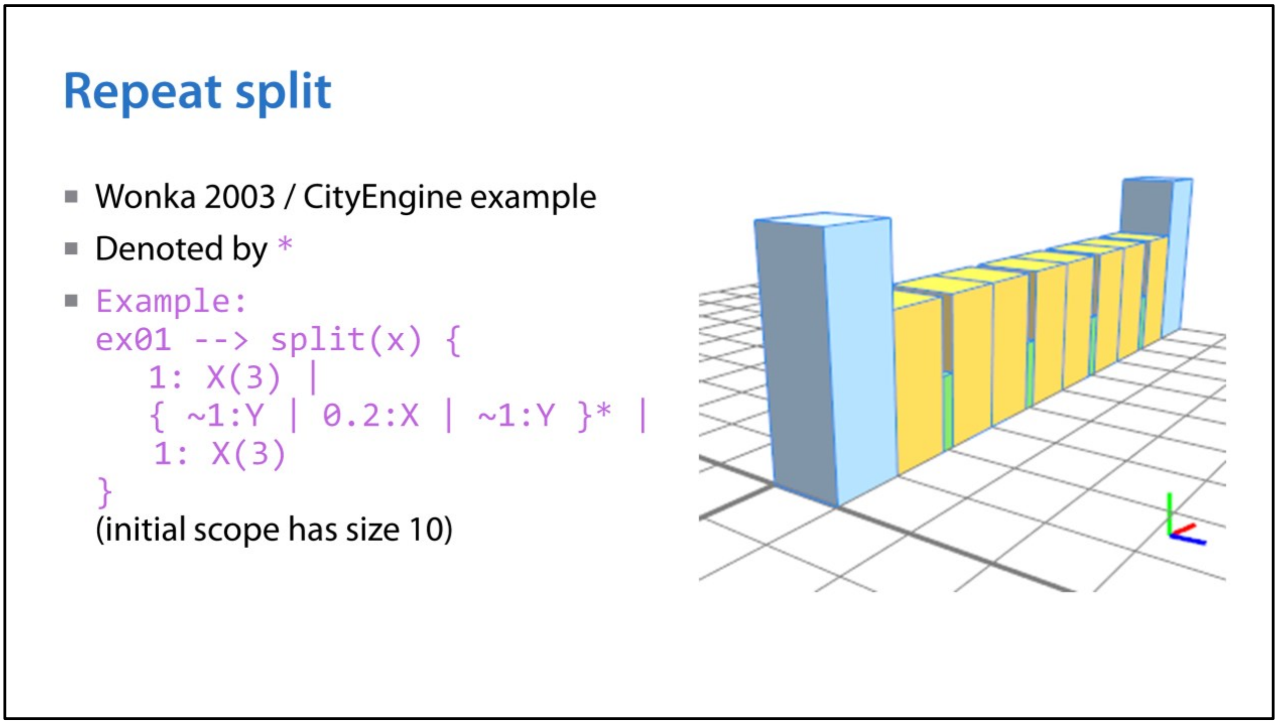
- First, the absolute values get priority from left to right, e.g.
  `split_example03 --> split(x)`
  `{ 2:X | 1:A | 1:Z | 2:Y | 1:Z }`
- Second, relative values (~) divide remaining space according to their weights
  `split_example03 --> split(x)`
  `{ 1.5:X | ~3:Y | 1.5:X | 0.5:X }`
- Not all specified shapes might be generated

Image: CityEngine Online Help System



Splitting design choices

- What splitting axis are allowed?
  - Axis-aligned splits
  - General 3D axis
  - General splits
- Is splitting of arbitrary geometry supported?
  - Splitting of 2D geometry (for lots)
  - Splitting of 3D geometry
  - Splitting of boxes only
- Grid Split vs. 1D split



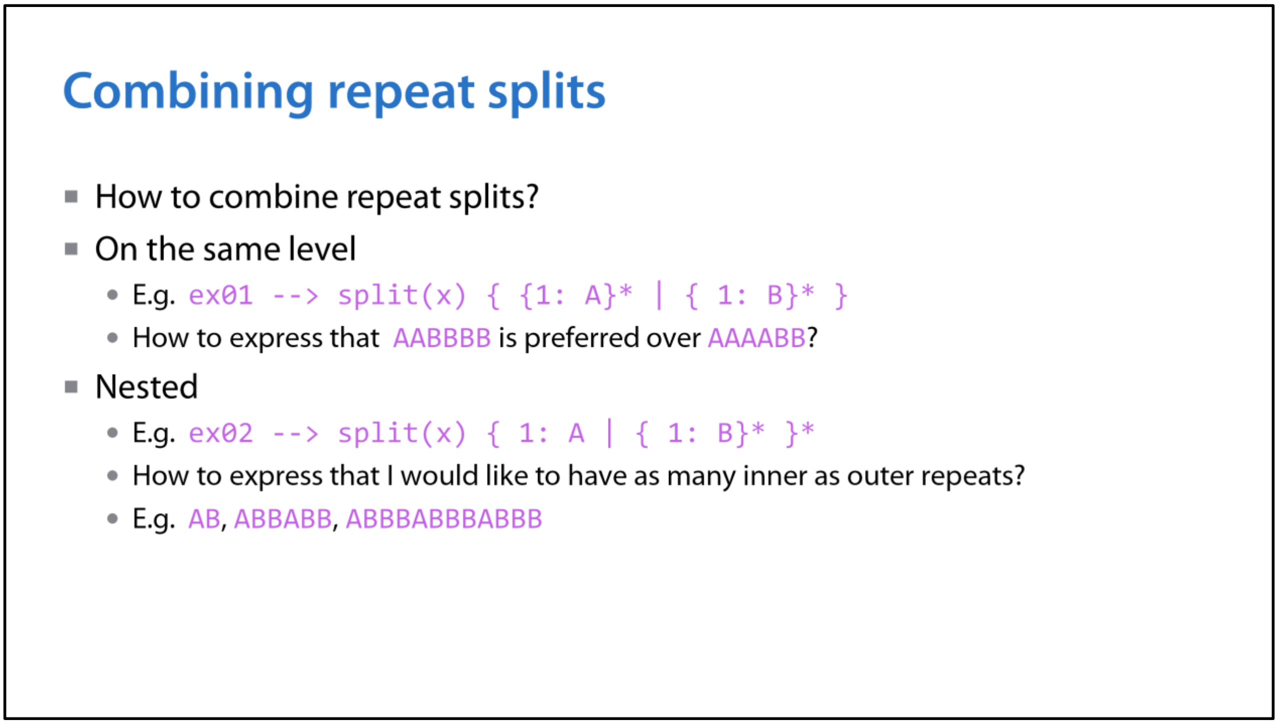
Repeat split

- Wonka 2003 / CityEngine example
- Denoted by *
- Example:

```
ex01 --> split(x) {
   1: X(3) |
   { ~1:Y | 0.2:X | ~1:Y }* |
    1: X(3)
}
```

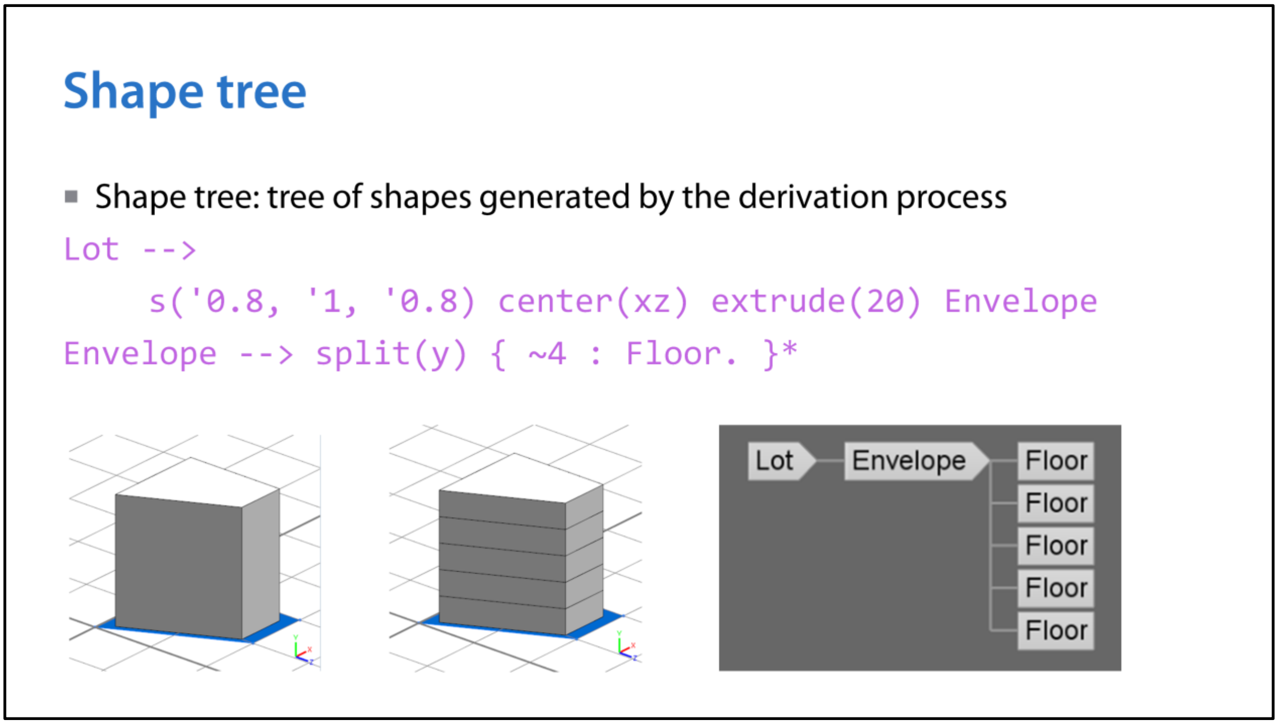
(initial scope has size 10)

Image: CityEngine Online Help System



Combining repeat splits

- How to combine repeat splits?
- On the same level
  - E.g. `ex01 --> split(x) { {1: A}* | { 1: B}* }`
  - How to express that AABBBB is preferred over AAAABB?
- Nested
  - E.g. `ex02 --> split(x) { 1: A | { 1: B}* }*`
  - How to express that I would like to have as many inner as outer repeats?
  - E.g. AB, ABBABB, ABBBABBBABBB



Shape tree

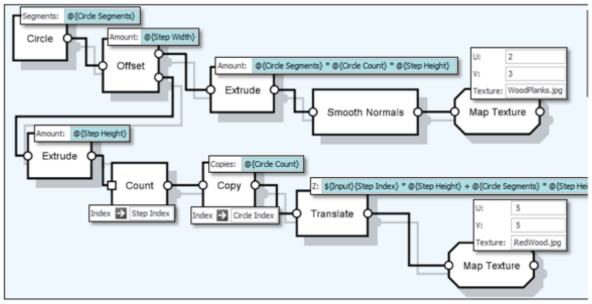
- Shape tree: tree of shapes generated by the derivation process

```
Lot -->
    s('0.8, '1, '0.8) center(xz) extrude(20) Envelope
Envelope --> split(y) { ~4 : Floor. }*
```

Image: CityEngine Online Help System

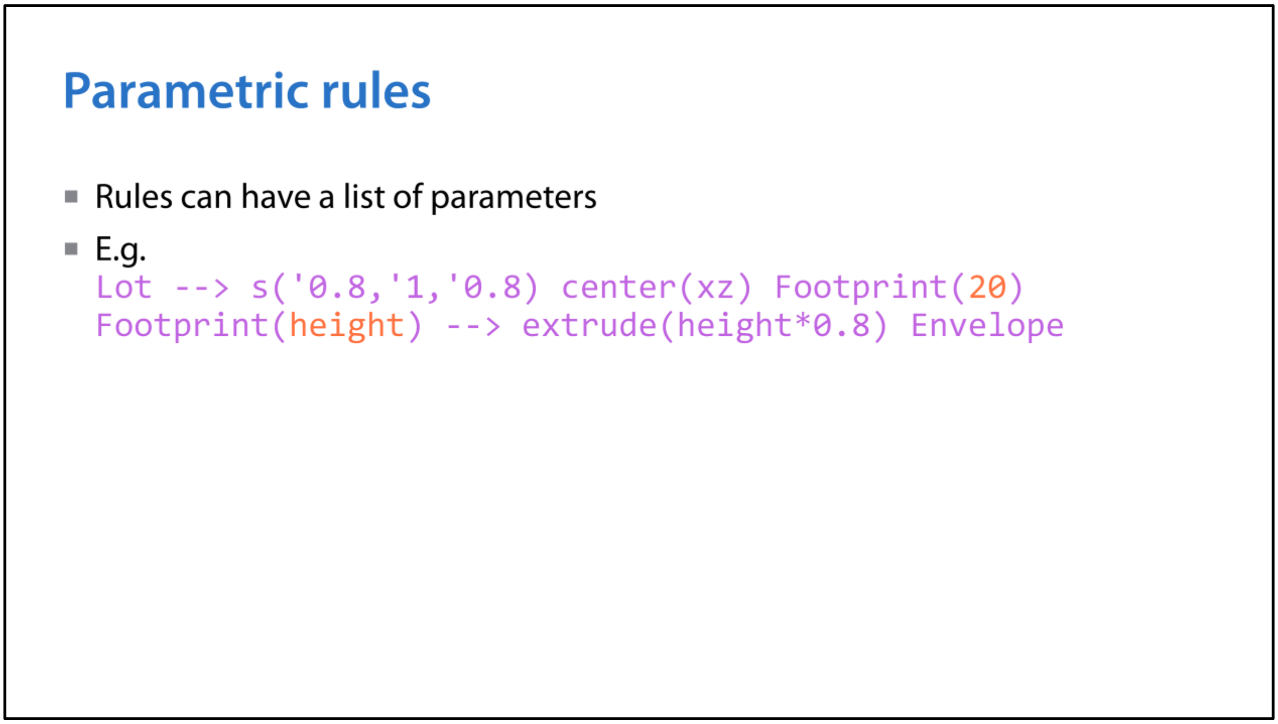
Graph-based representations

- Shape operations are represented as nodes
- Nodes have inputs and outputs
- Data flow is controlled by edges
- Examples: Silva et al., Thaller et al., Patow, Houdini



Segments: @{Circle Segments}
Amount: @{Step Width}
Amount: @{Circle Segments} * @{Circle Count} * @{Step Height}
U: 2
V: 3
Texture: WoodPlanks.jpg
Amount: @{Step Height}
Copies: @{Circle Count}
Index → Step Index
Index → Circle Index
Z: ${Input}{Step Index} * @{Step Height} + @{Circle Segments} * @{Step Hei
U: 5
V: 5
Texture: RedWood.jpg

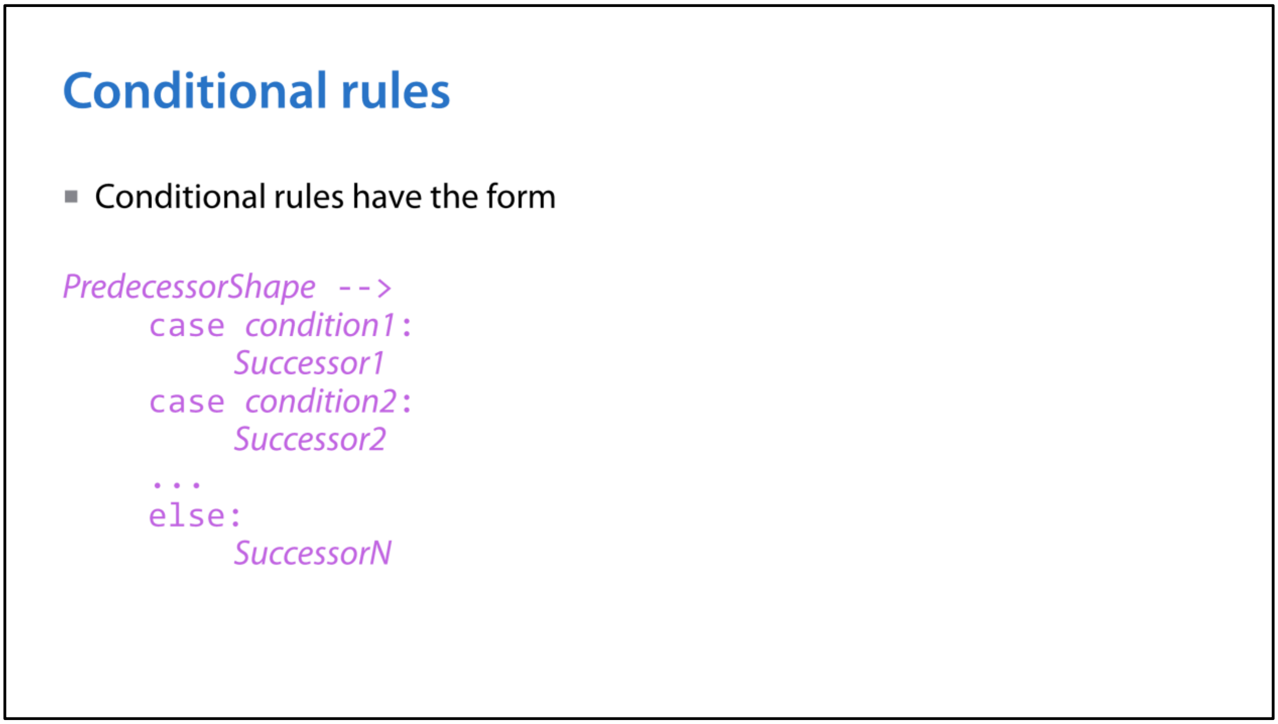
Image: Silva 2015



Parametric rules

- Rules can have a list of parameters
- E.g.

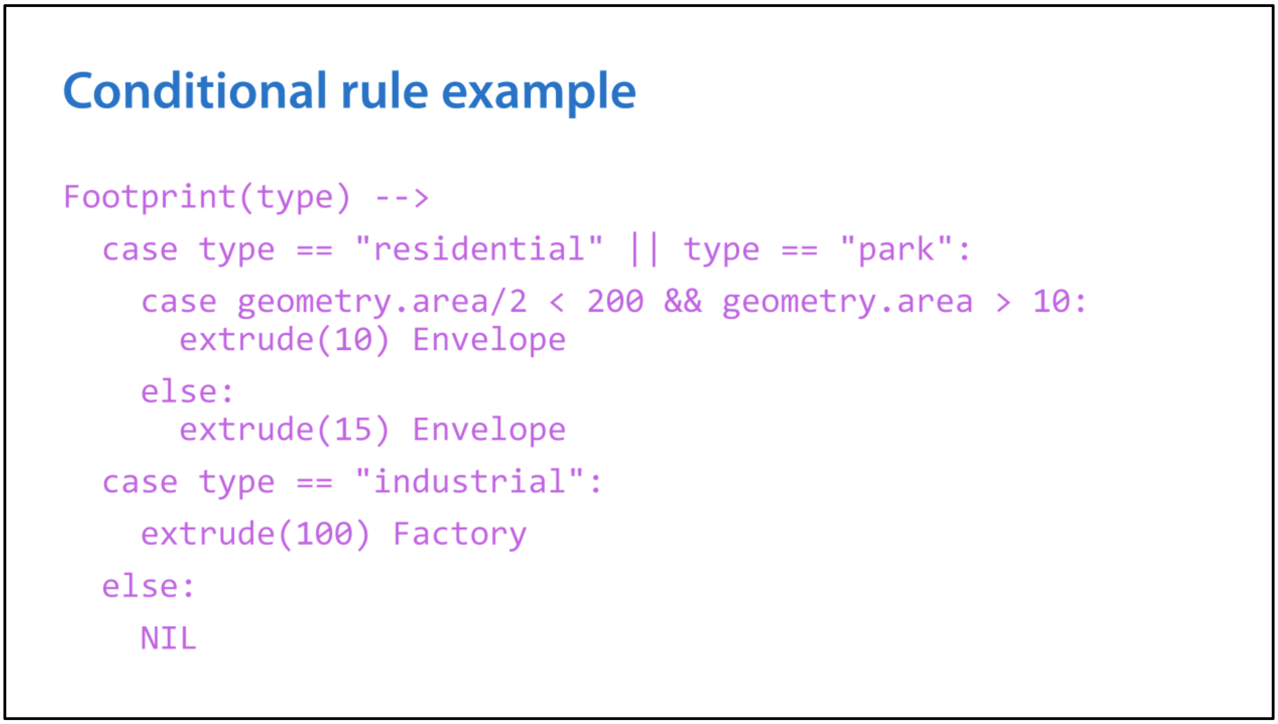
```
Lot --> s('0.8,'1,'0.8) center(xz) Footprint(20)
Footprint(height) --> extrude(height*0.8) Envelope
```



Conditional rules

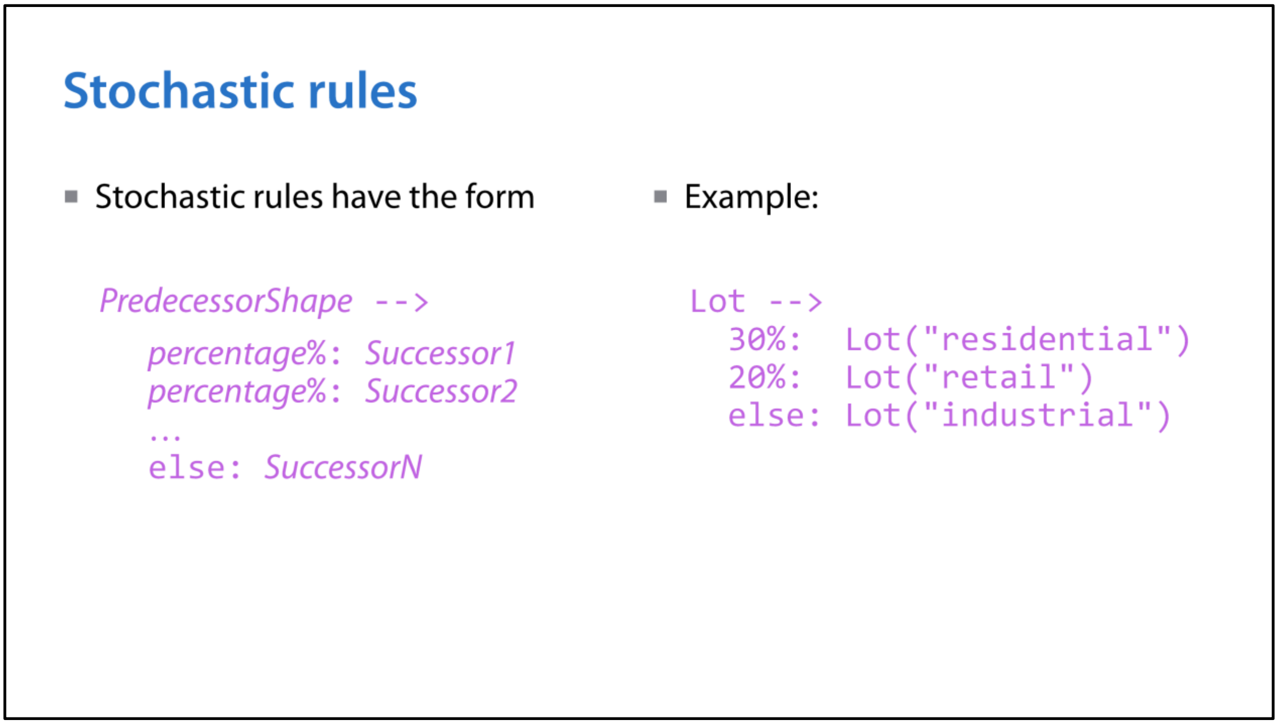
- Conditional rules have the form

```
PredecessorShape -->
     case condition1:
          Successor1
     case condition2:
          Successor2
     ...
     else:
          SuccessorN
```



Conditional rule example

```
Footprint(type) -->
  case type == "residential" || type == "park":
    case geometry.area/2 < 200 && geometry.area > 10:
      extrude(10) Envelope
    else:
      extrude(15) Envelope
  case type == "industrial":
    extrude(100) Factory
  else:
    NIL
```



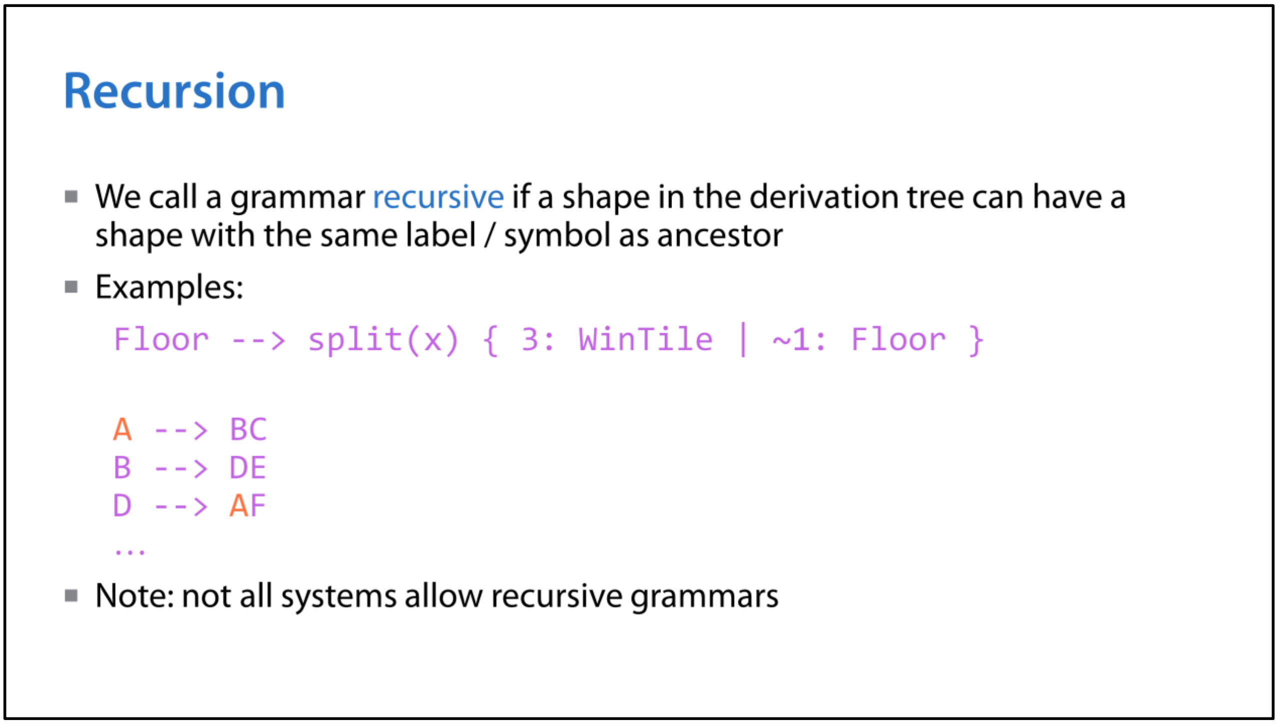
Stochastic rules

- Stochastic rules have the form

```
PredecessorShape -->
    percentage%:  Successor1
    percentage%:  Successor2
    ...
    else:  SuccessorN
```

- Example:

```
Lot -->
  30%:   Lot("residential")
  20%:   Lot("retail")
  else:  Lot("industrial")
```

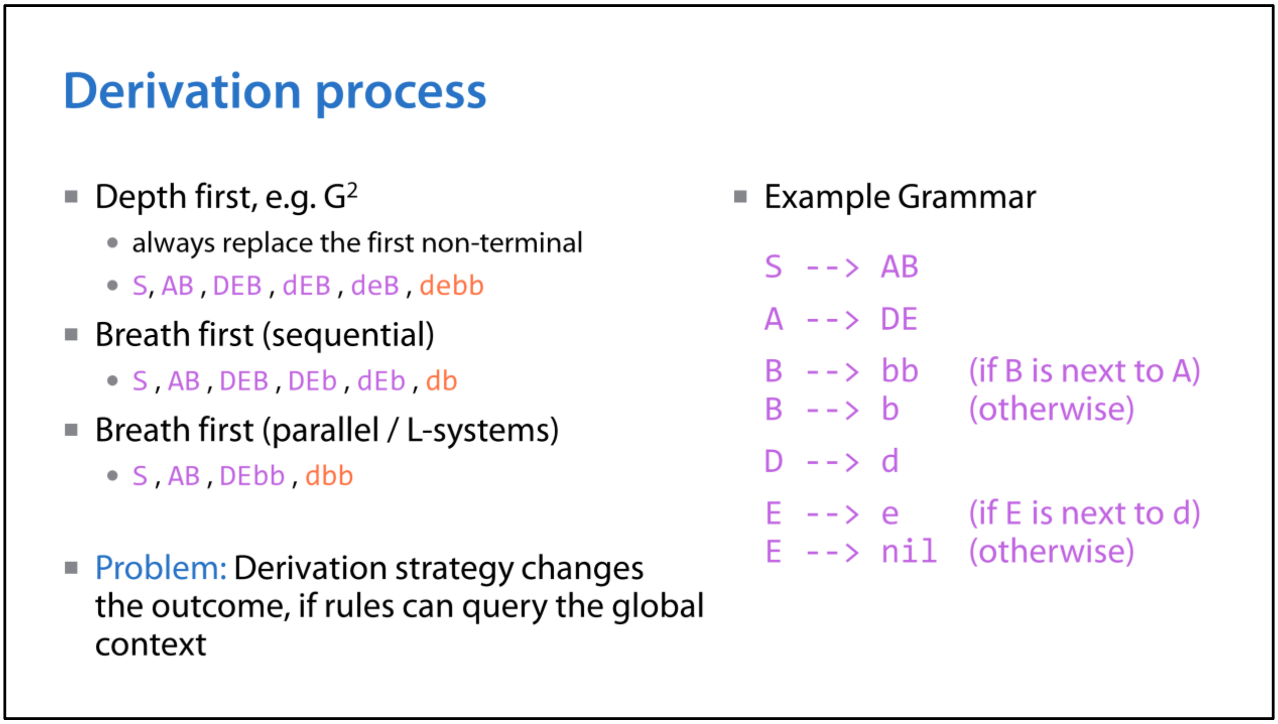


Recursion

- We call a grammar recursive if a shape in the derivation tree can have a shape with the same label / symbol as ancestor
- Examples:

```
Floor --> split(x) { 3: WinTile | ~1: Floor }

A --> BC
B --> DE
D --> AF
...
```

- Note: not all systems allow recursive grammars

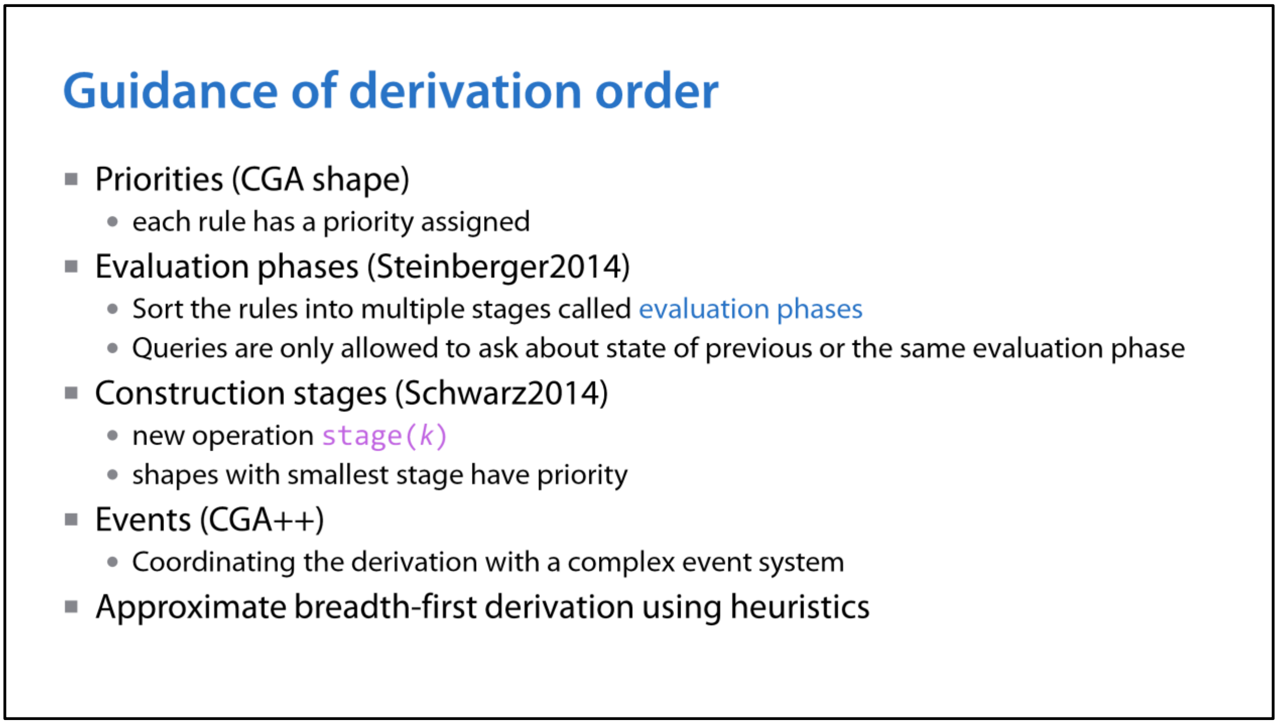


Derivation process

- Depth first, e.g. G^2
  - always replace the first non-terminal
  - S, AB , DEB , dEB , deB , debb
- Breath first (sequential)
  - S , AB , DEB , DEb , dEb , db
- Breath first (parallel / L-systems)
  - S , AB , DEbb , dbb
- Problem: Derivation strategy changes the outcome, if rules can query the global context

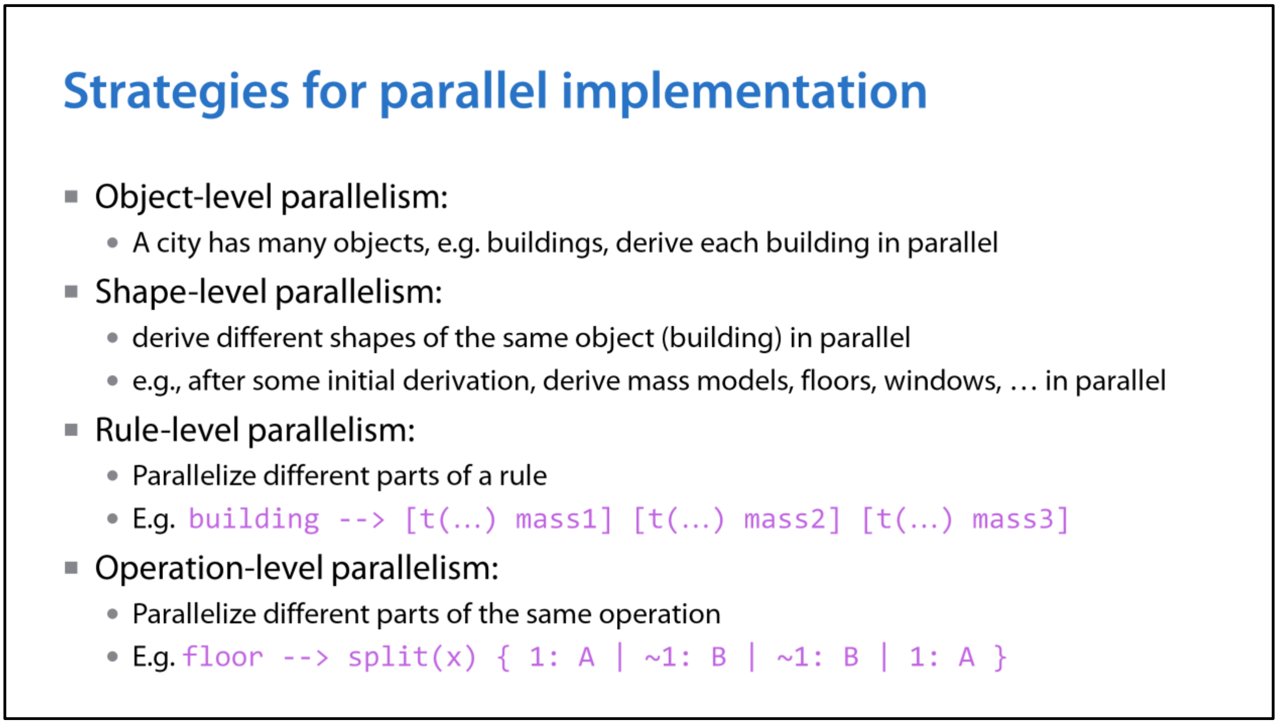
- Example Grammar

```
S --> AB
A --> DE
B --> bb     (if B is next to A)
B --> b      (otherwise)
D --> d
E --> e      (if E is next to d)
E --> nil    (otherwise)
```



Guidance of derivation order

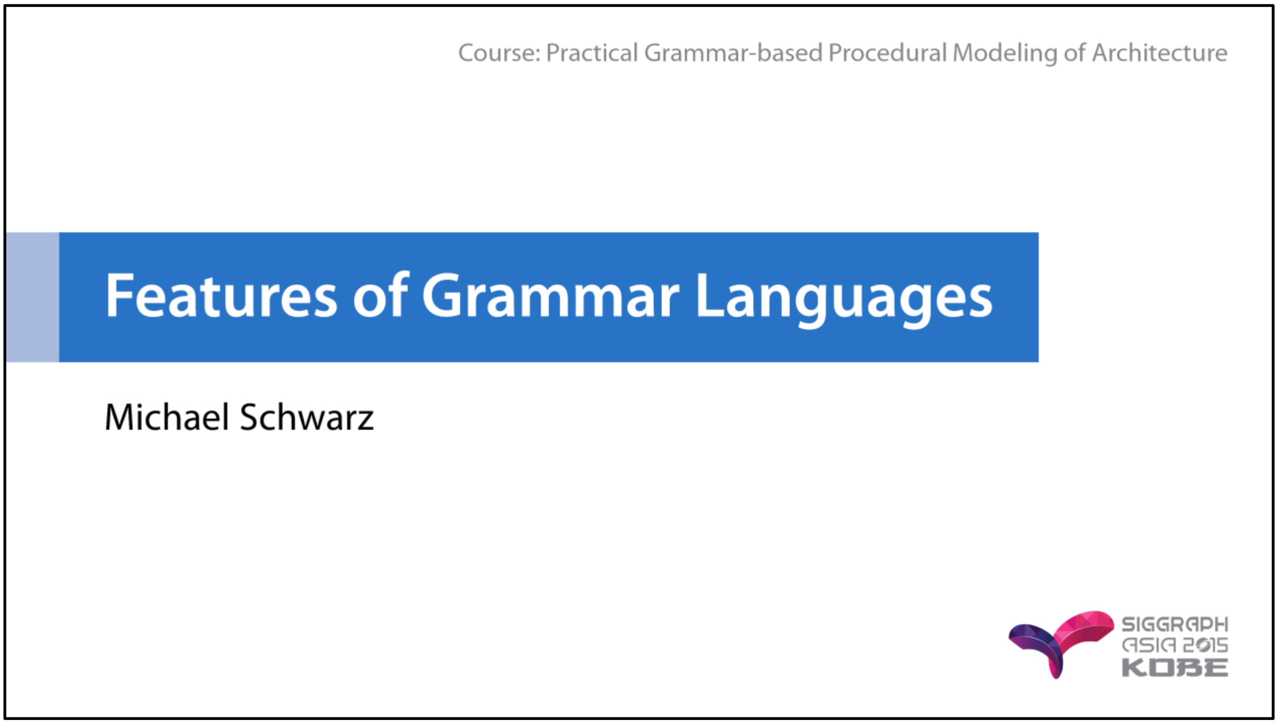
- Priorities (CGA shape)
  - each rule has a priority assigned
- Evaluation phases (Steinberger2014)
  - Sort the rules into multiple stages called evaluation phases
  - Queries are only allowed to ask about state of previous or the same evaluation phase
- Construction stages (Schwarz2014)
  - new operation `stage(k)`
  - shapes with smallest stage have priority
- Events (CGA++)
  - Coordinating the derivation with a complex event system
- Approximate breadth-first derivation using heuristics



Strategies for parallel implementation

- Object-level parallelism:
  - A city has many objects, e.g. buildings, derive each building in parallel
- Shape-level parallelism:
  - derive different shapes of the same object (building) in parallel
  - e.g., after some initial derivation, derive mass models, floors, windows, … in parallel
- Rule-level parallelism:
  - Parallelize different parts of a rule
  - E.g. `building --> [t(…) mass1] [t(…) mass2] [t(…) mass3]`
- Operation-level parallelism:
  - Parallelize different parts of the same operation
  - E.g. `floor --> split(x) { 1: A | ~1: B | ~1: B | 1: A }`

3 Features of grammar languages

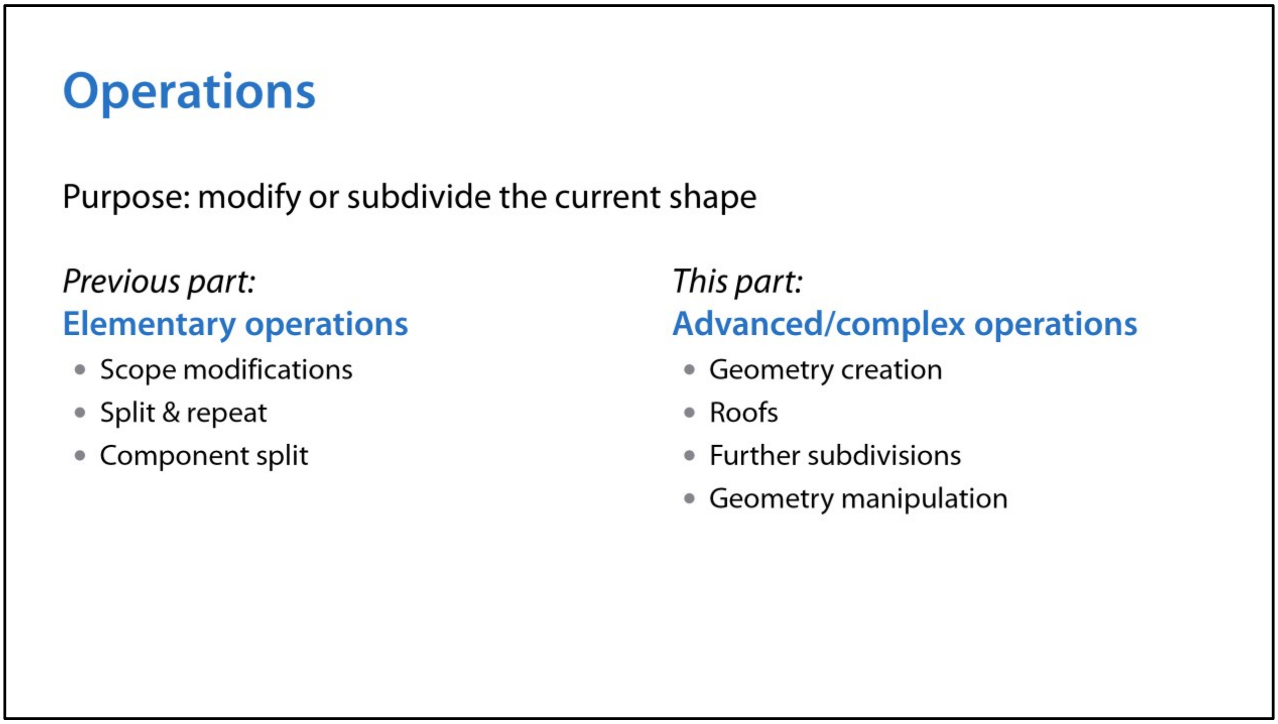


Course: Practical Grammar-based Procedural Modeling of Architecture

Features of Grammar Languages

Michael Schwarz

SIGGRAPH ASIA 2015 KOBE



Operations

Purpose: modify or subdivide the current shape

Previous part:

Elementary operations

- Scope modifications
- Split & repeat
- Component split

This part:

Advanced/complex operations

- Geometry creation
- Roofs
- Further subdivisions
- Geometry manipulation

Geometry creation

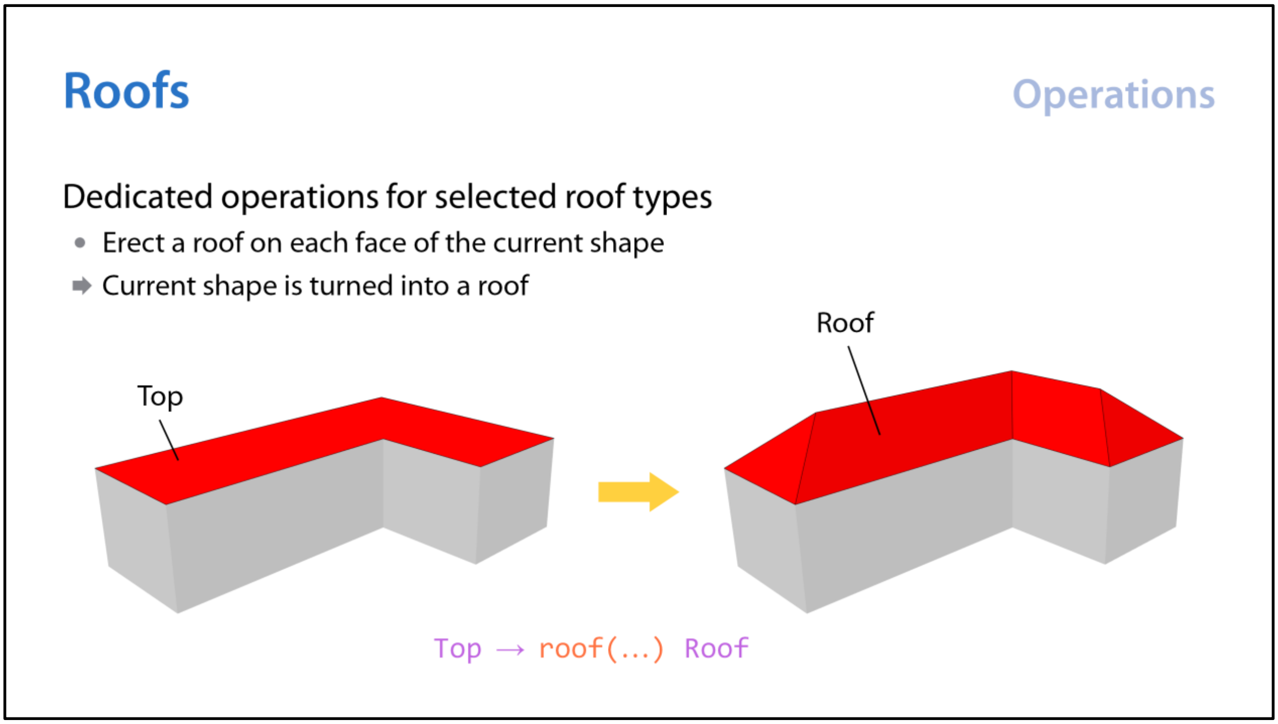
Operations

Create new shape geometry

- Usually based on current shape geometry

Examples

- Create pre-defined shape (e.g., circle)
- Load geometry from asset
- Explicit constructors
- Extrusion: `extrude(`*amount*`)`
- Find inscribed rectangles (e.g., `innerRect` (CE))
- Erect roofs



Roofs

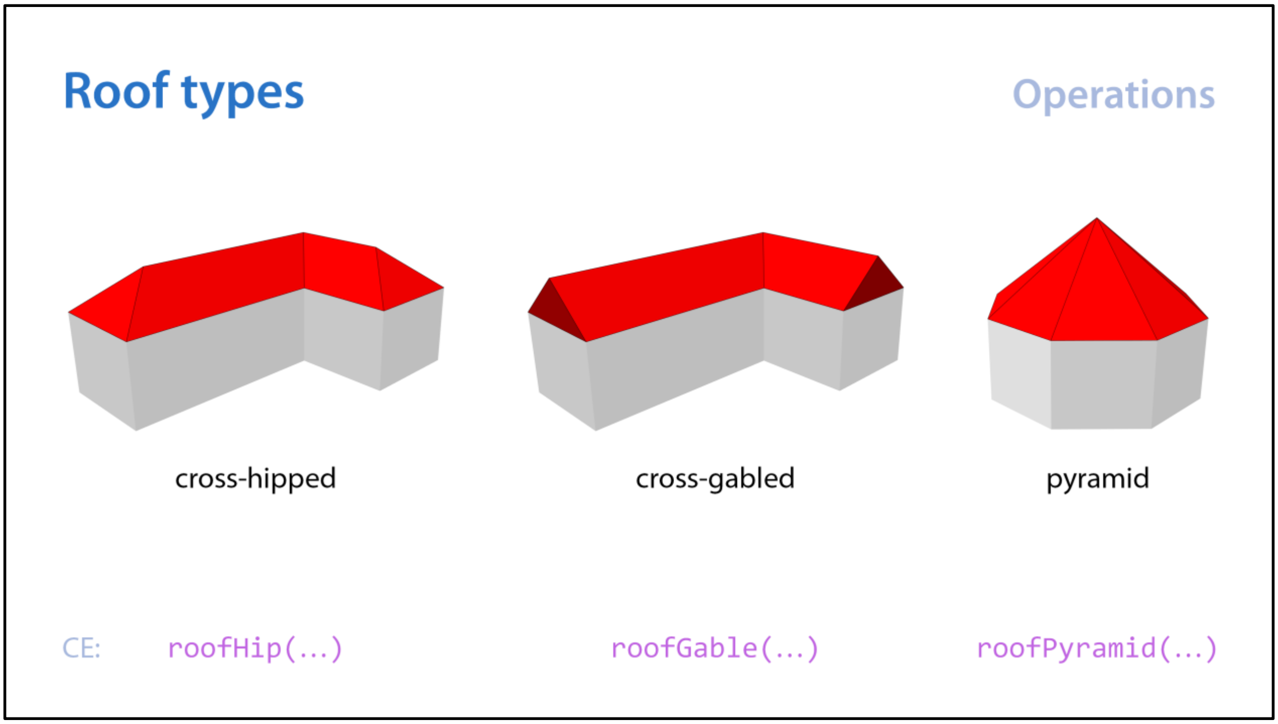
Operations

Dedicated operations for selected roof types

- Erect a roof on each face of the current shape

➡ Current shape is turned into a roof

`Top → roof(...) Roof`



Roof types

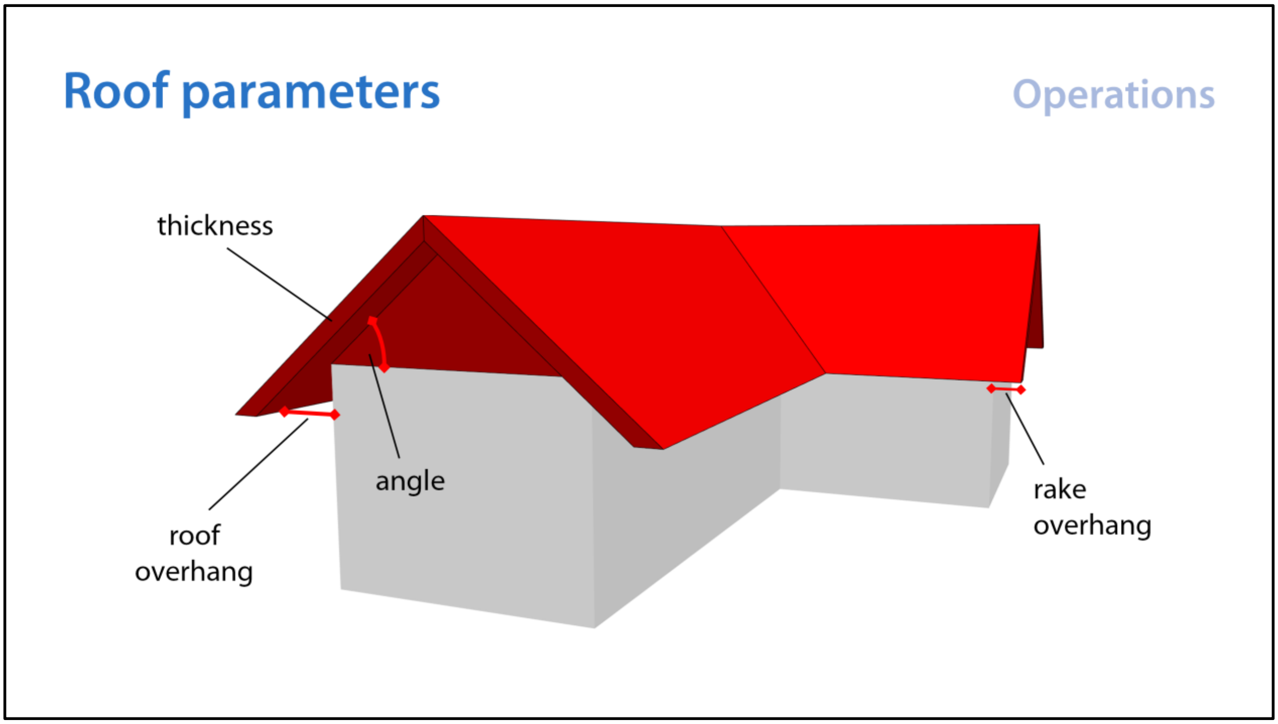
Operations

cross-hipped

cross-gabled

pyramid

CE: `roofHip(...)` `roofGable(...)` `roofPyramid(...)`

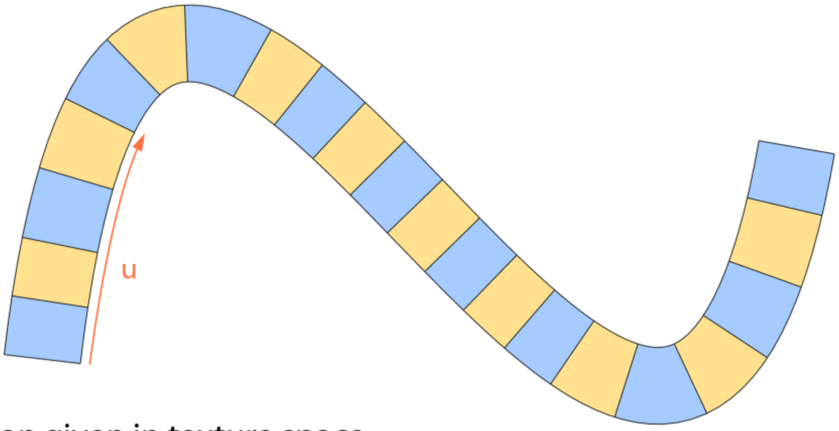


Roof parameters

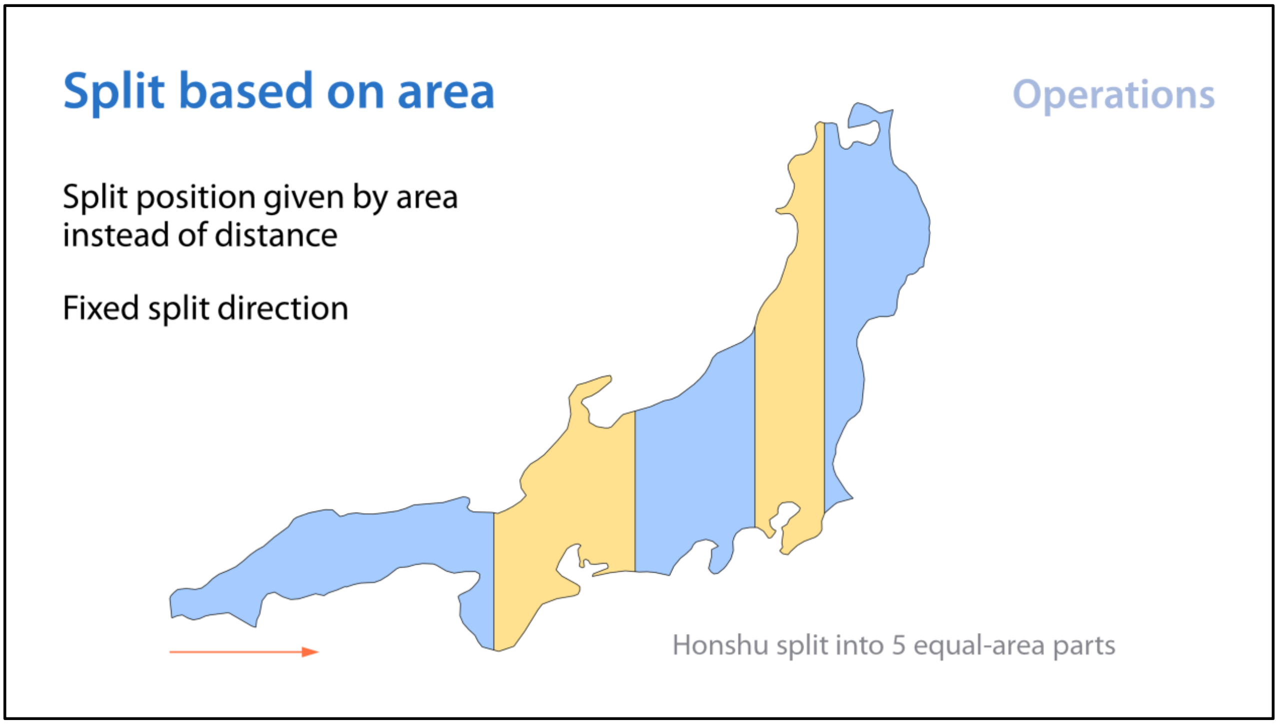
Operations

Split based on texture coordinates

Operations



Split position given in texture space



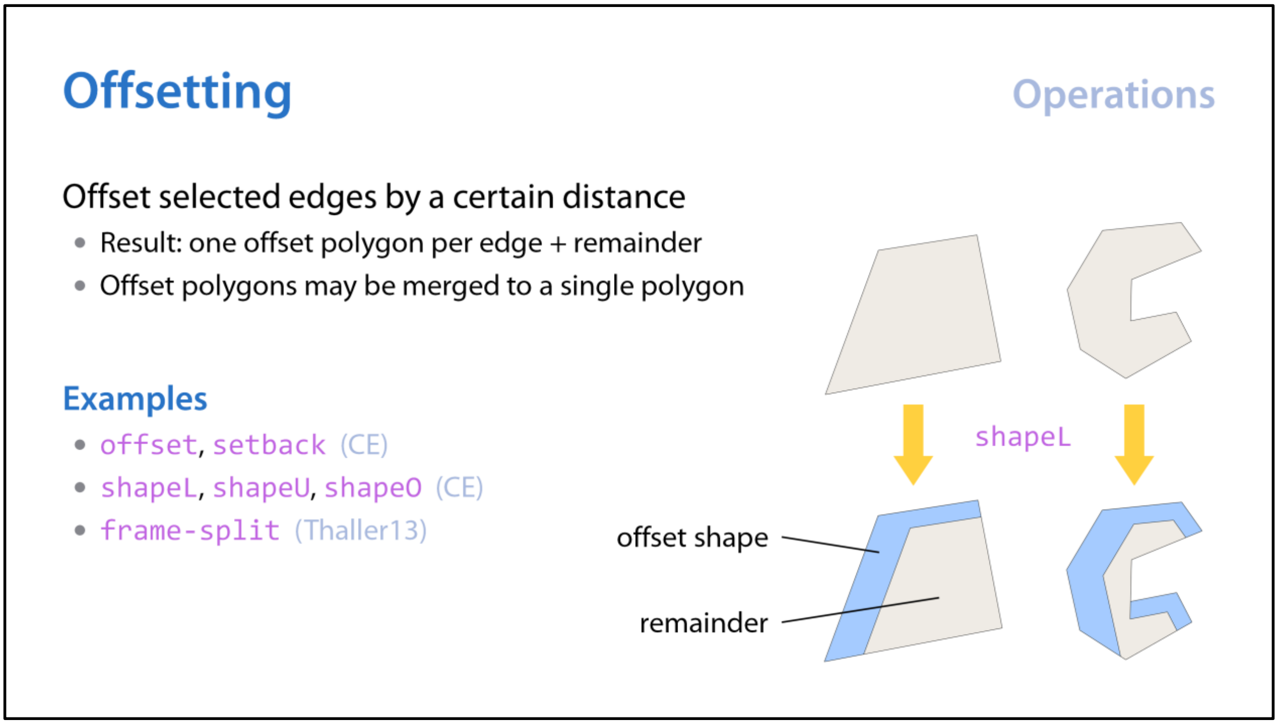
Split based on area

Operations

Split position given by area instead of distance

Fixed split direction

Honshu split into 5 equal-area parts



Offsetting

Operations

Offset selected edges by a certain distance

- Result: one offset polygon per edge + remainder
- Offset polygons may be merged to a single polygon

Examples

- `offset`, `setback` (CE)
- `shapeL`, `shapeU`, `shapeO` (CE)
- `frame-split` (Thaller13)

frame-split: uniform inward offset, partitioning similar to straight-skeleton-based approach

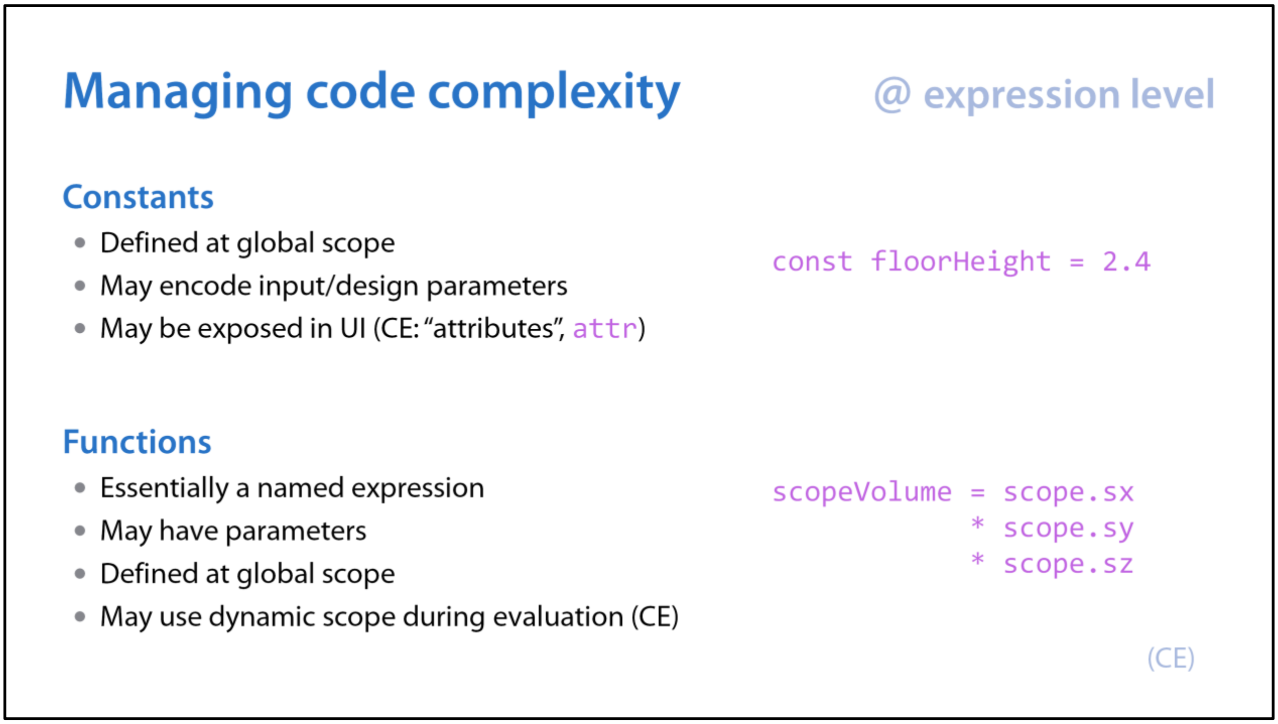
Geometry manipulation

Operations

Manipulate/transform current shape geometry

Examples

- Reverse normals
- Remove collinear vertices
- Remove holes
- Split non-convex faces into convex parts
- Simplify geometry for lower level of detail
- Compute or transform texture coordinates



Managing code complexity

@ expression level

Constants

- Defined at global scope
- May encode input/design parameters
- May be exposed in UI (CE: "attributes", `attr`)

```
const floorHeight = 2.4
```

Functions

- Essentially a named expression
- May have parameters
- Defined at global scope
- May use dynamic scope during evaluation (CE)

```
scopeVolume = scope.sx
            * scope.sy
            * scope.sz
```

(CE)

Managing code complexity

@ rule level

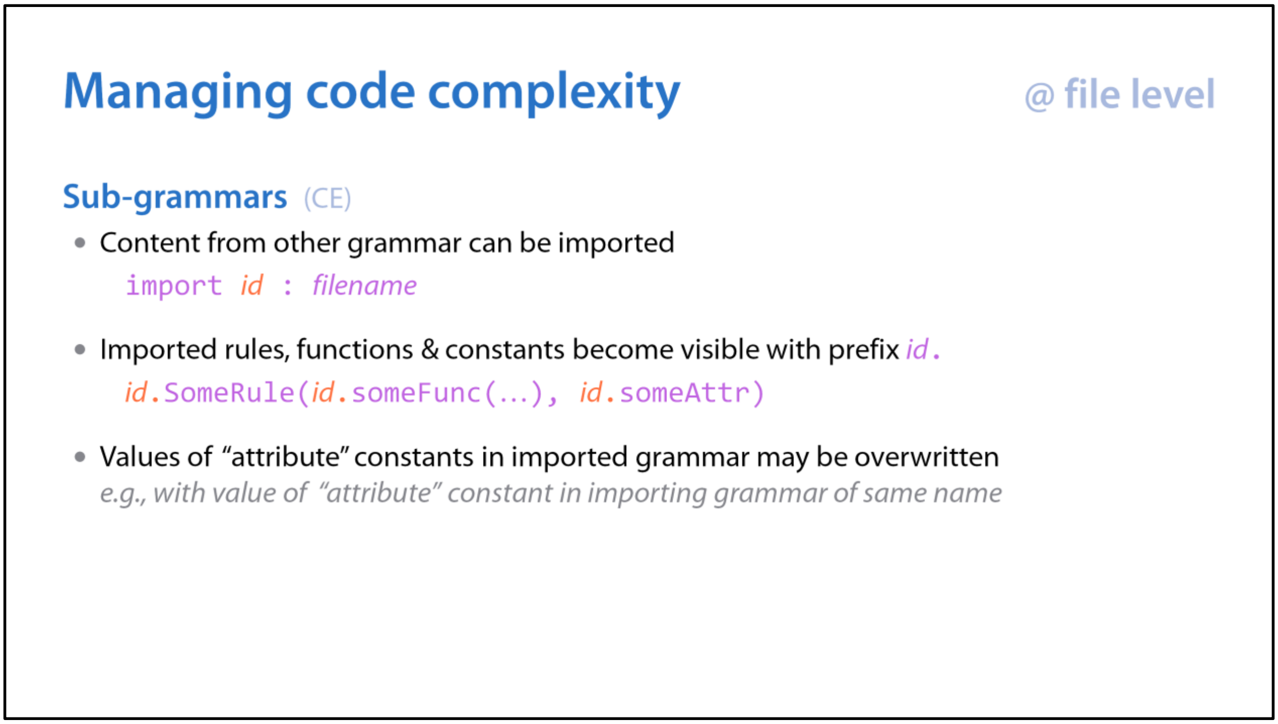
Modules (G²)

- Rule with sub-rules
- Parameters of rule are accessible by sub-rule
- Sub-rules live in new namespace

- Rule prefixes

`:`	Sub-rule
`_`	Global rule
`../`	Sub-rule of parent module

```
$Rule:Box(w:Float, h:Float)
  -> repeatX(w, :$SubRule1);
{
  $SubRule1:Box
    -> repeatY(h, $SubRule2);
  $SubRule2:Box
    -> ...
}
```



Managing code complexity

@ file level

Sub-grammars (CE)

- Content from other grammar can be imported

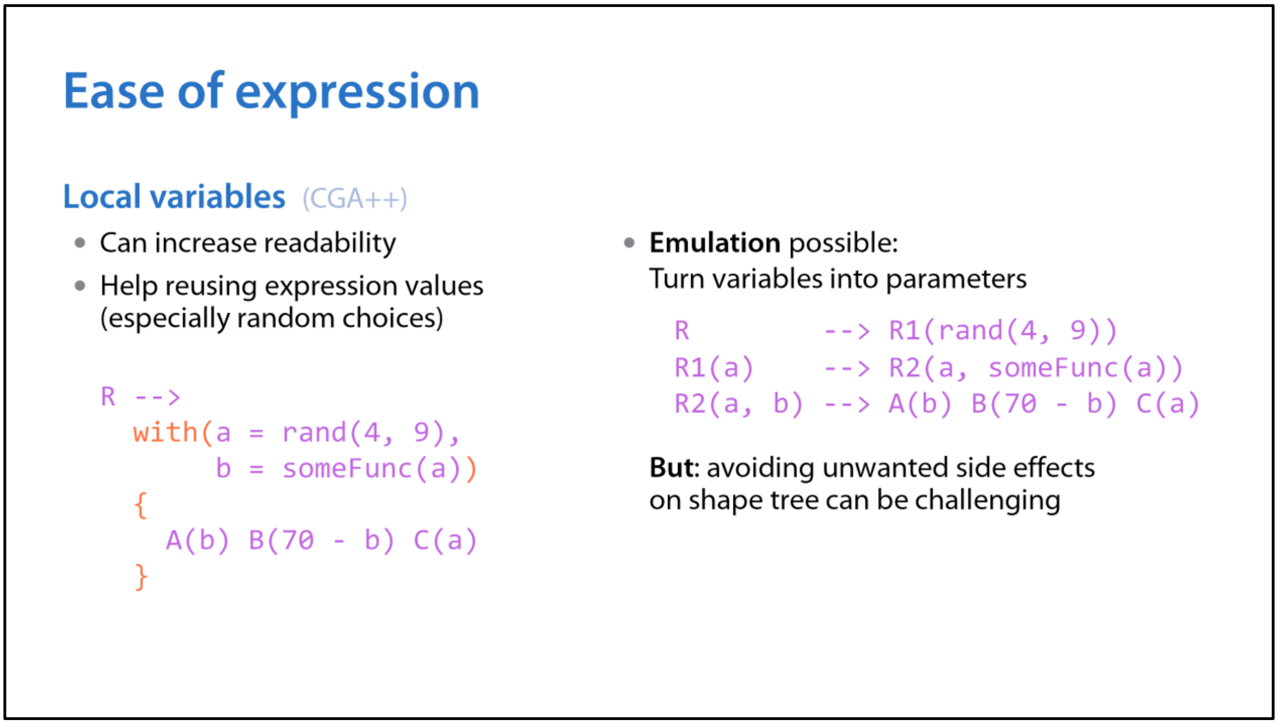
  `import` *id* `:` *filename*

- Imported rules, functions & constants become visible with prefix *id*`.`

  id`.SomeRule(`*id*`.someFunc(…),` *id*`.someAttr)`

- Values of "attribute" constants in imported grammar may be overwritten

  e.g., with value of "attribute" constant in importing grammar of same name



Ease of expression

Local variables (CGA++)

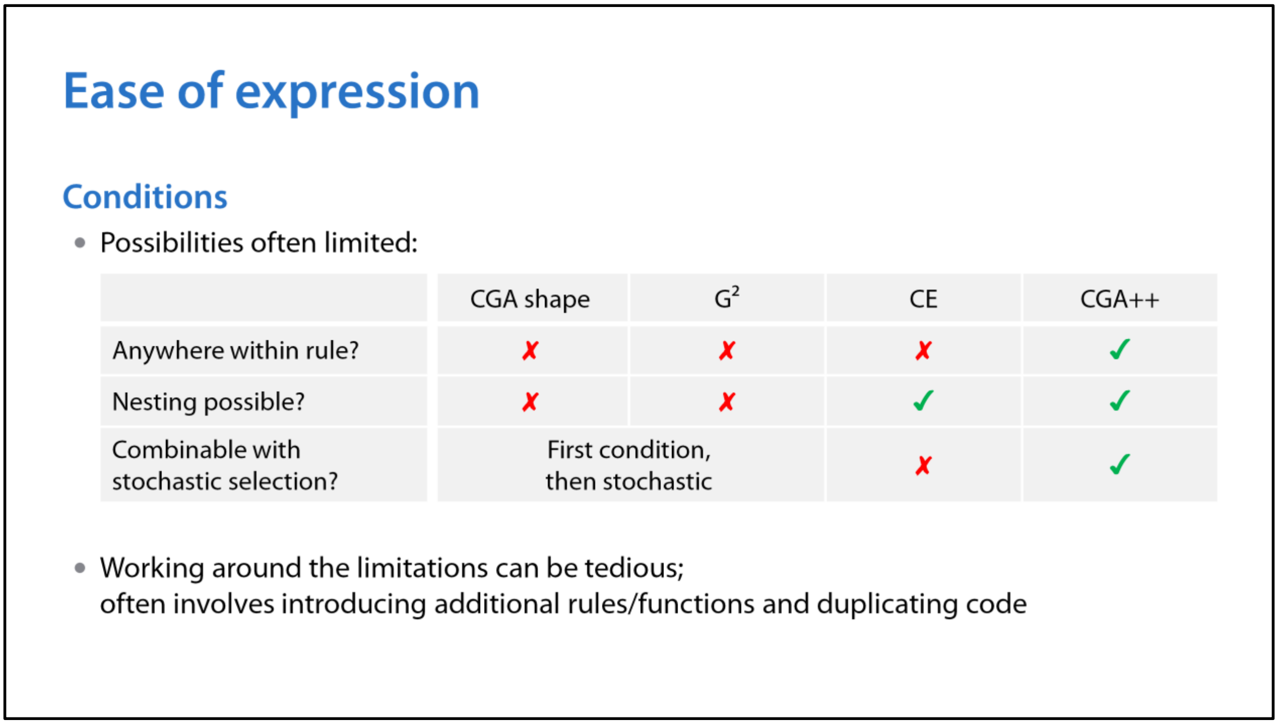
- Can increase readability
- Help reusing expression values (especially random choices)

```
R -->
  with(a = rand(4, 9),
       b = someFunc(a))
  {
    A(b) B(70 - b) C(a)
  }
```

- **Emulation** possible:
  Turn variables into parameters

```
R         --> R1(rand(4, 9))
R1(a)     --> R2(a, someFunc(a))
R2(a, b) --> A(b) B(70 - b) C(a)
```

But: avoiding unwanted side effects on shape tree can be challenging



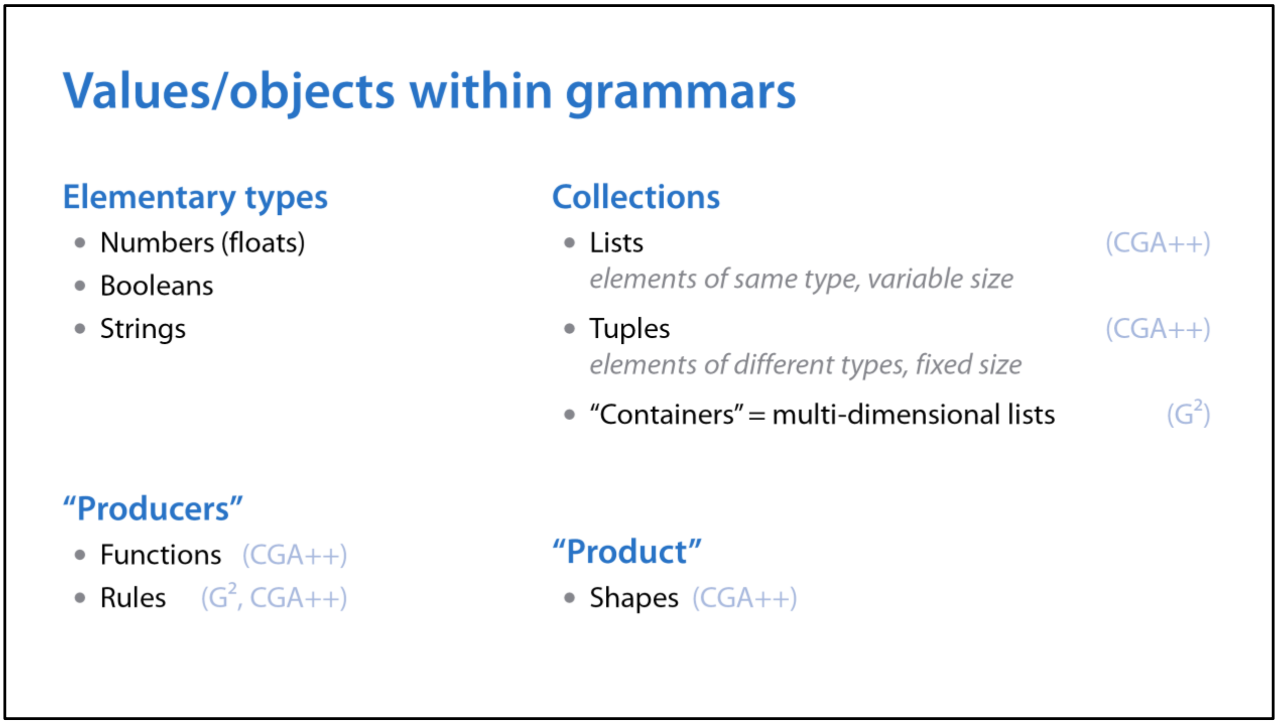
Ease of expression

Conditions

- Possibilities often limited:

	CGA shape	G^2	CE	CGA++
Anywhere within rule?	✗	✗	✗	✓
Nesting possible?	✗	✗	✓	✓
Combinable with stochastic selection?	First condition, then stochastic		✗	✓

- Working around the limitations can be tedious;
  often involves introducing additional rules/functions and duplicating code



Values/objects within grammars

Elementary types

- Numbers (floats)
- Booleans
- Strings

Collections

- Lists (CGA++)
  elements of same type, variable size
- Tuples (CGA++)
  elements of different types, fixed size
- "Containers" = multi-dimensional lists (G^2)

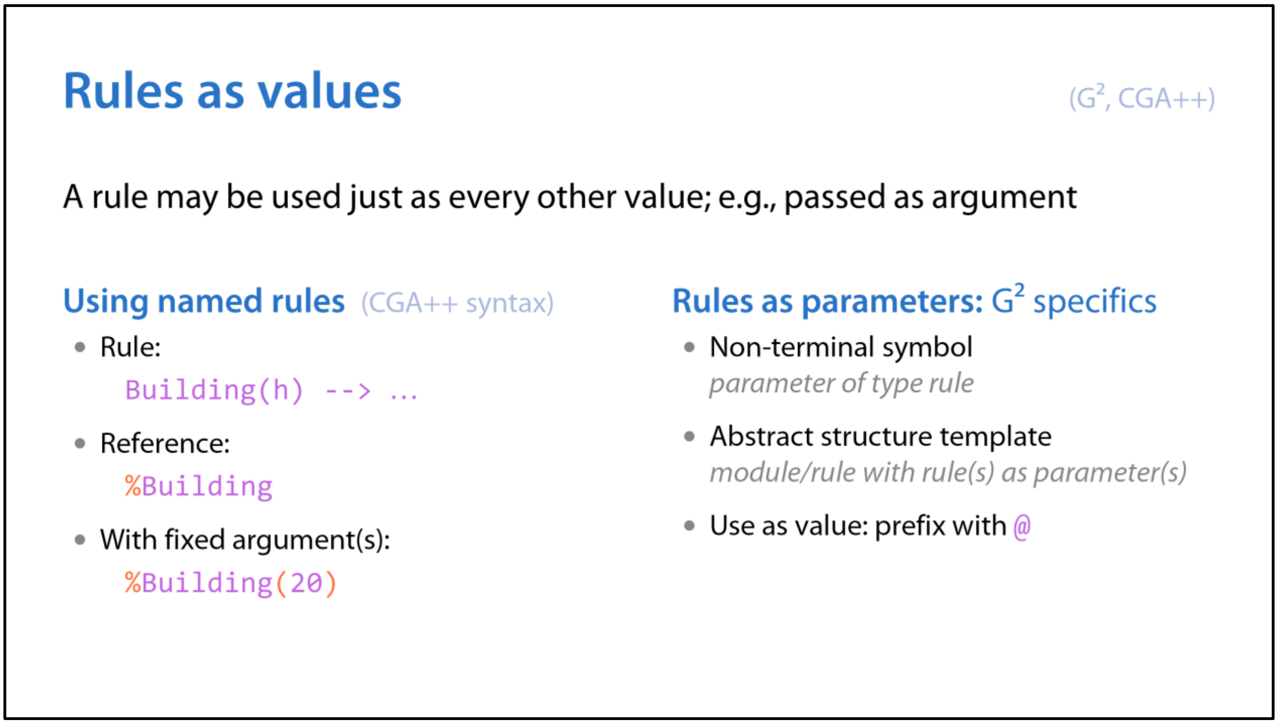
"Producers"

- Functions (CGA++)
- Rules (G^2, CGA++)

"Product"

- Shapes (CGA++)

CE has functions for representing a list as a string, where elements are separated by a semicolon.



Rules as values

(G², CGA++)

A rule may be used just as every other value; e.g., passed as argument

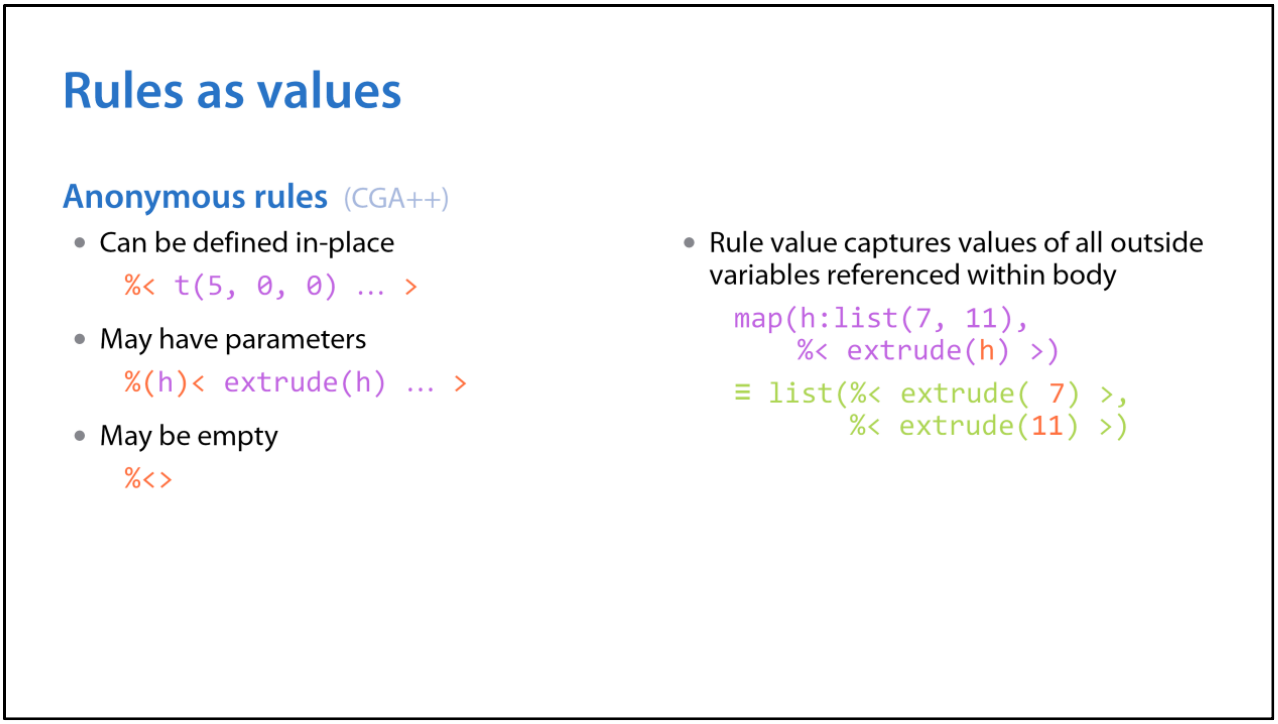
Using named rules (CGA++ syntax)

- Rule:
  `Building(h) --> ...`
- Reference:
  `%Building`
- With fixed argument(s):
  `%Building(20)`

Rules as parameters: G² specifics

- Non-terminal symbol
  parameter of type rule
- Abstract structure template
  module/rule with rule(s) as parameter(s)
- Use as value: prefix with `@`

At least in CGA++, rules are full first-class citizens and hence cannot only be passed around but may also be stored in collections or as shape attributes.



Rules as values

Anonymous rules (CGA++)

- Can be defined in-place

```
%< t(5, 0, 0) … >
```

- May have parameters

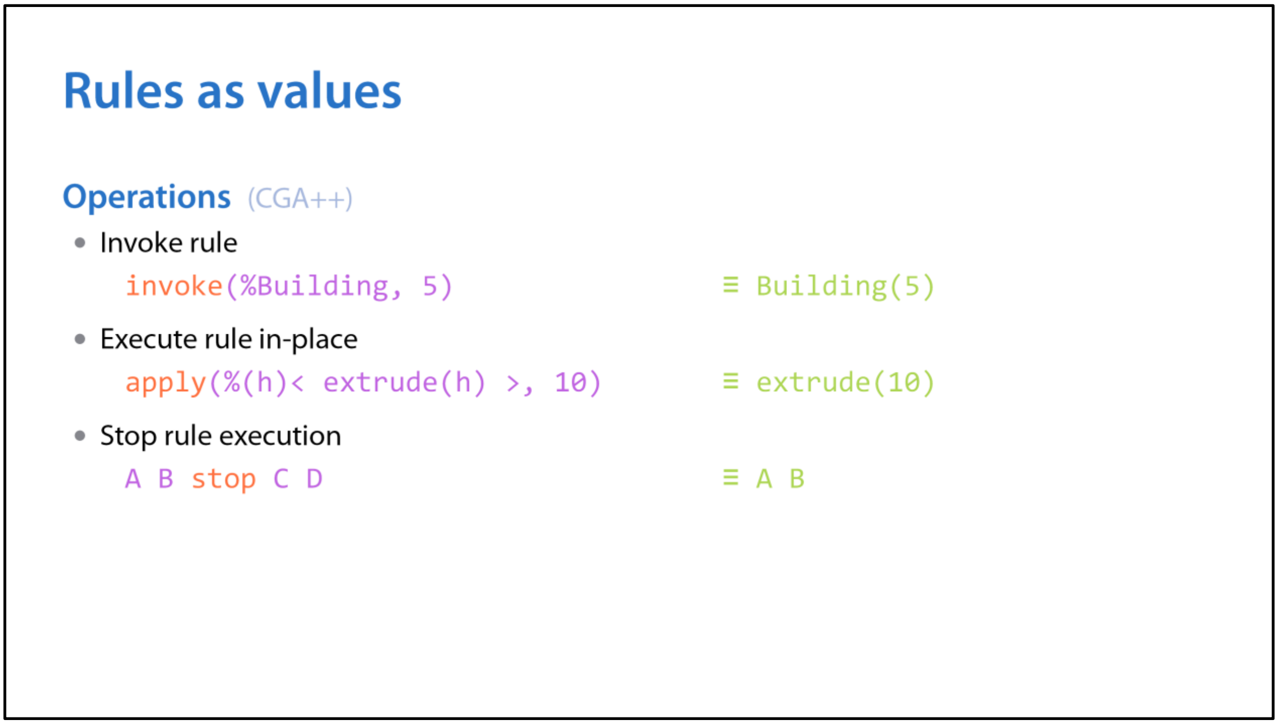
```
%(h)< extrude(h) … >
```

- May be empty

```
%<>
```

- Rule value captures values of all outside variables referenced within body

```
map(h:list(7, 11),
    %< extrude(h) >)
≡ list(%< extrude( 7) >,
       %< extrude(11) >)
```



Rules as values

Operations (CGA++)

- Invoke rule

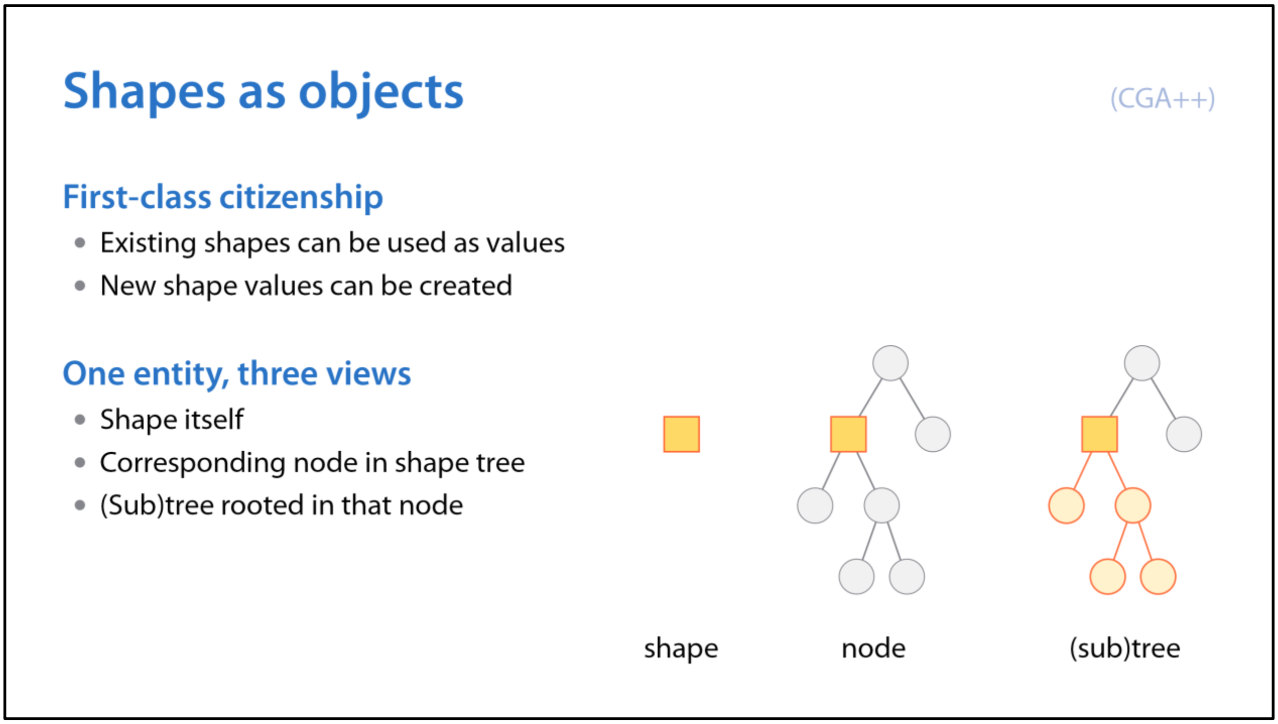
  `invoke(%Building, 5)` ≡ `Building(5)`

- Execute rule in-place

  `apply(%(h)< extrude(h) >, 10)` ≡ `extrude(10)`

- Stop rule execution

  `A B stop C D` ≡ `A B`



Shapes as objects

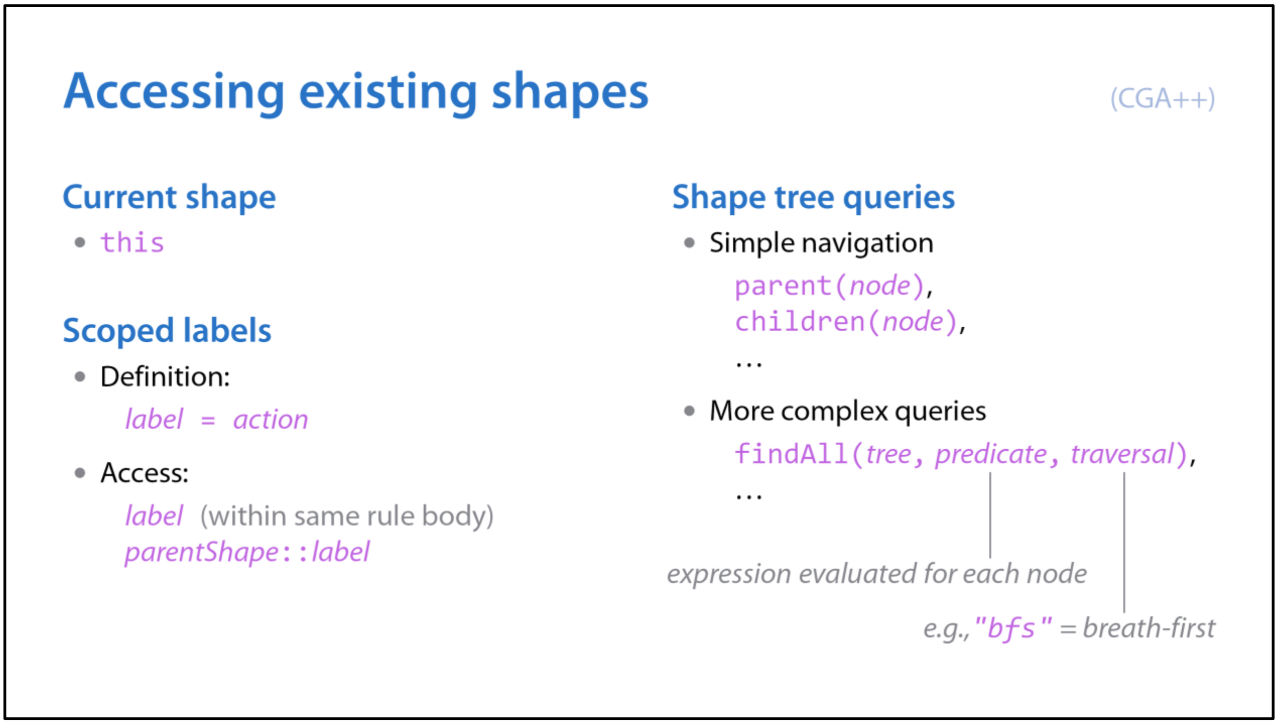
(CGA++)

First-class citizenship

- Existing shapes can be used as values
- New shape values can be created

One entity, three views

- Shape itself
- Corresponding node in shape tree
- (Sub)tree rooted in that node



Accessing existing shapes

(CGA++)

Current shape

- `this`

Scoped labels

- Definition:

  label = *action*

- Access:

  label (within same rule body)
  parentShape::*label*

Shape tree queries

- Simple navigation

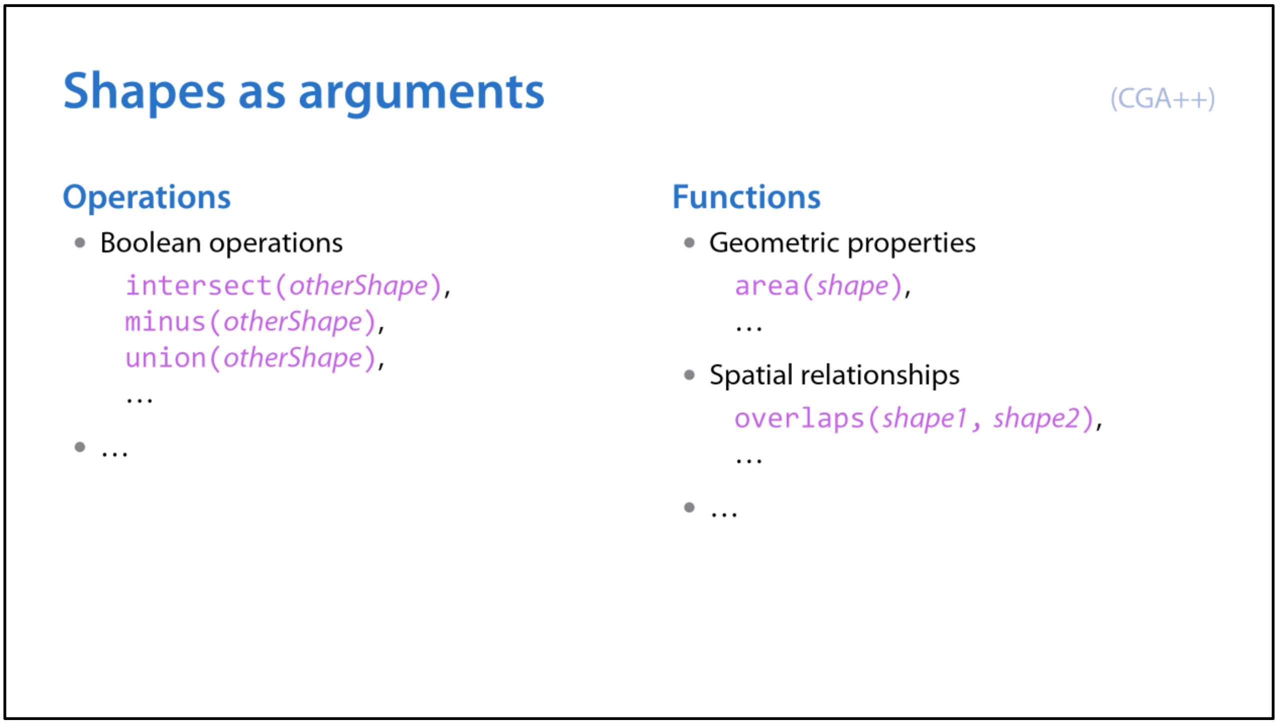
  parent(*node*),
  children(*node*),
  …

- More complex queries

  findAll(*tree*, *predicate*, *traversal*),
  …

expression evaluated for each node

e.g., "bfs" *= breath-first*



Shapes as arguments

(CGA++)

Operations

- Boolean operations

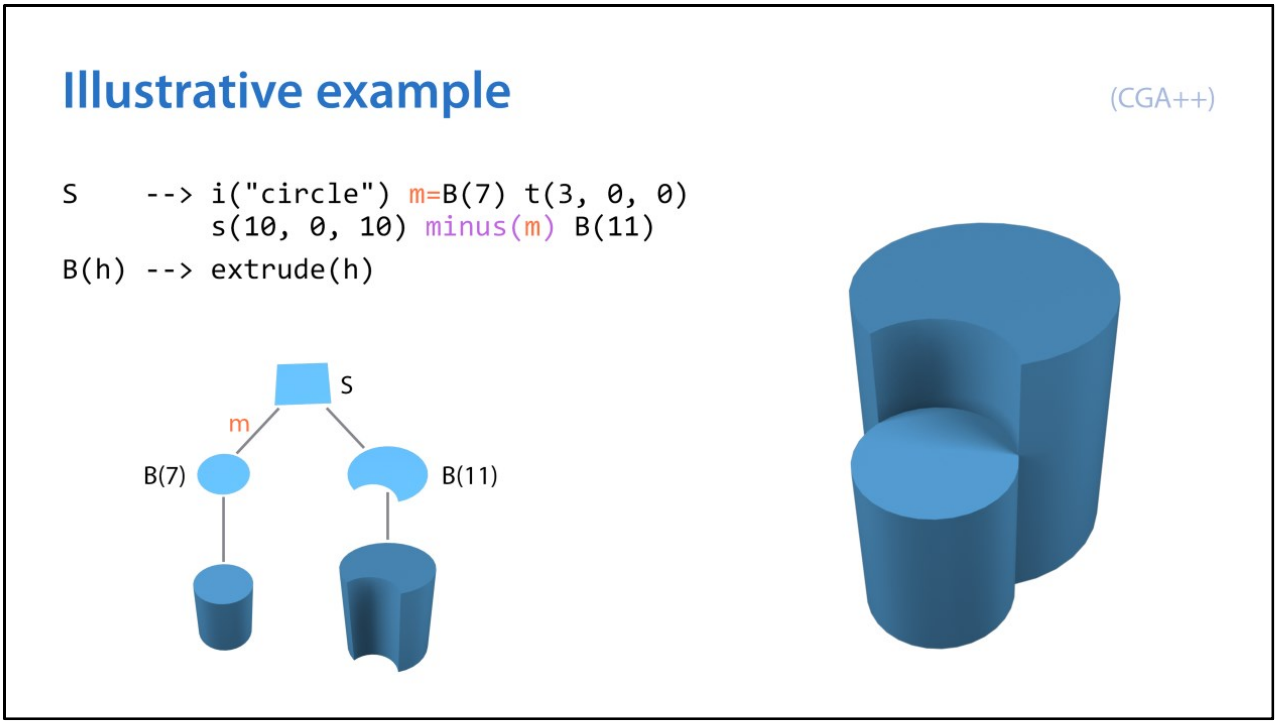
  `intersect(otherShape)`,
  `minus(otherShape)`,
  `union(otherShape)`,
  …
- …

Functions

- Geometric properties

  `area(shape)`,
  …
- Spatial relationships

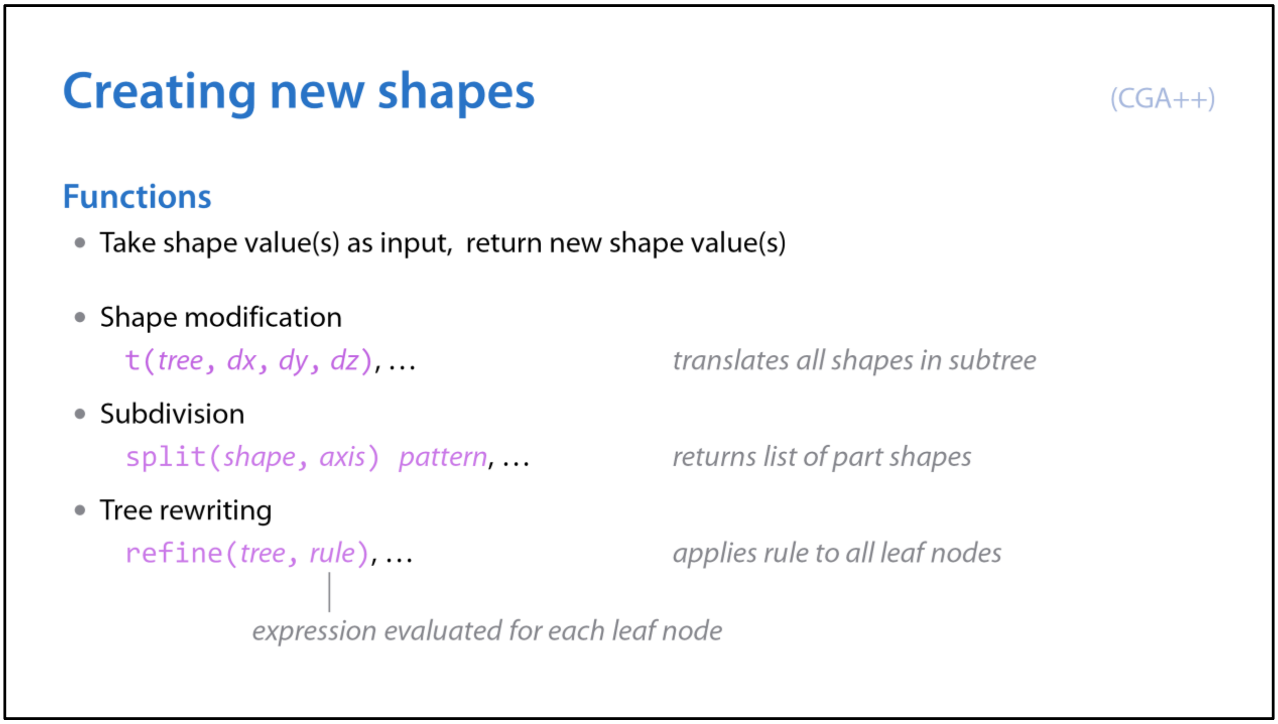
  `overlaps(shape1, shape2)`,
  …
- …



Illustrative example

(CGA++)

```
S    --> i("circle") m=B(7) t(3, 0, 0)
         s(10, 0, 10) minus(m) B(11)
B(h) --> extrude(h)
```



Creating new shapes

(CGA++)

Functions

- Take shape value(s) as input, return new shape value(s)

- Shape modification

  `t(`*tree*`,` *dx*`,` *dy*`,` *dz*`)`, … *translates all shapes in subtree*

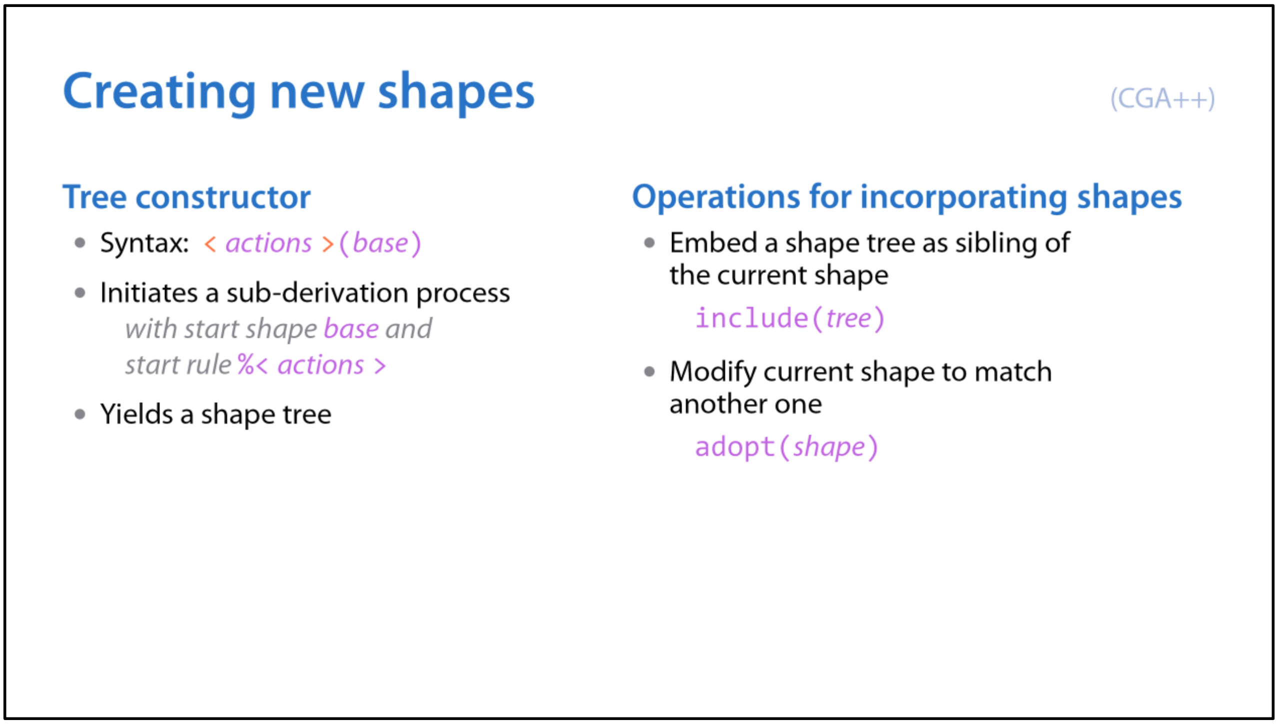
- Subdivision

  `split(`*shape*`,` *axis*`)` *pattern*, … *returns list of part shapes*

- Tree rewriting

  `refine(`*tree*`,` *rule*`)`, … *applies rule to all leaf nodes*

  expression evaluated for each leaf node



Creating new shapes

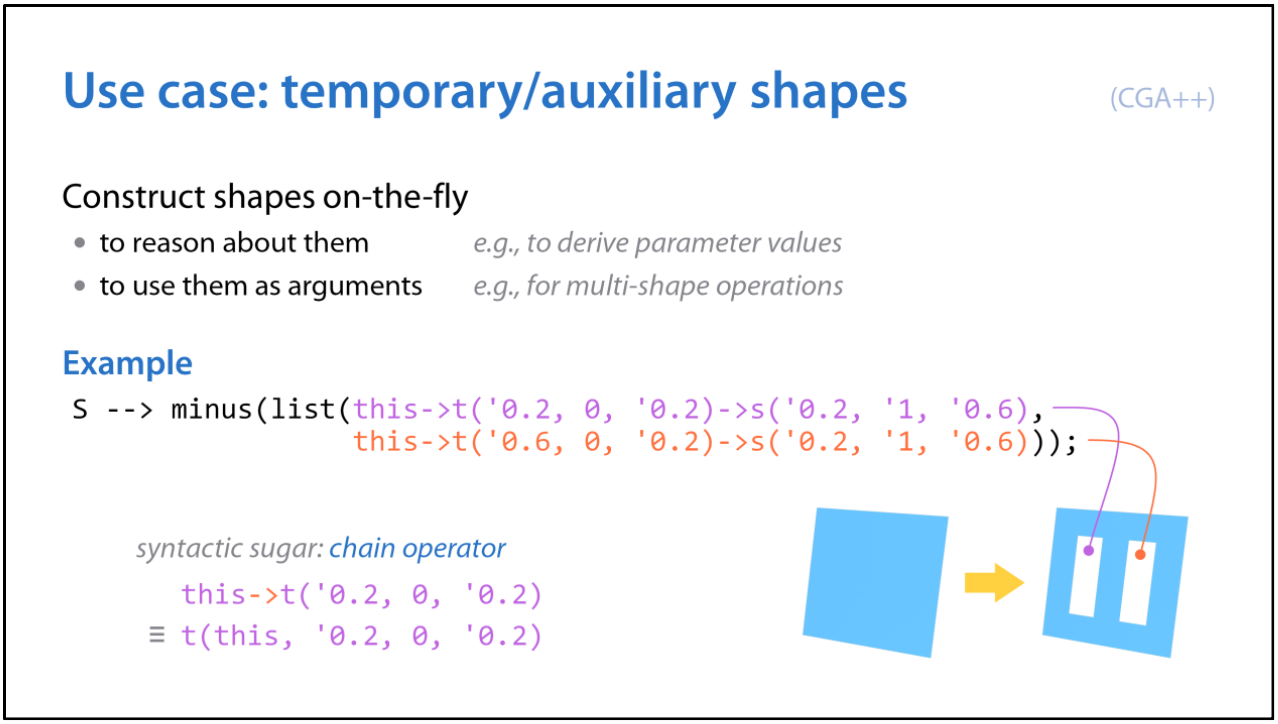
(CGA++)

Tree constructor

- Syntax: < *actions* >(*base*)
- Initiates a sub-derivation process
  with start shape base and
  start rule %< actions >
- Yields a shape tree

Operations for incorporating shapes

- Embed a shape tree as sibling of the current shape
  `include(`*tree*`)`
- Modify current shape to match another one
  `adopt(`*shape*`)`



Use case: temporary/auxiliary shapes

(CGA++)

Construct shapes on-the-fly

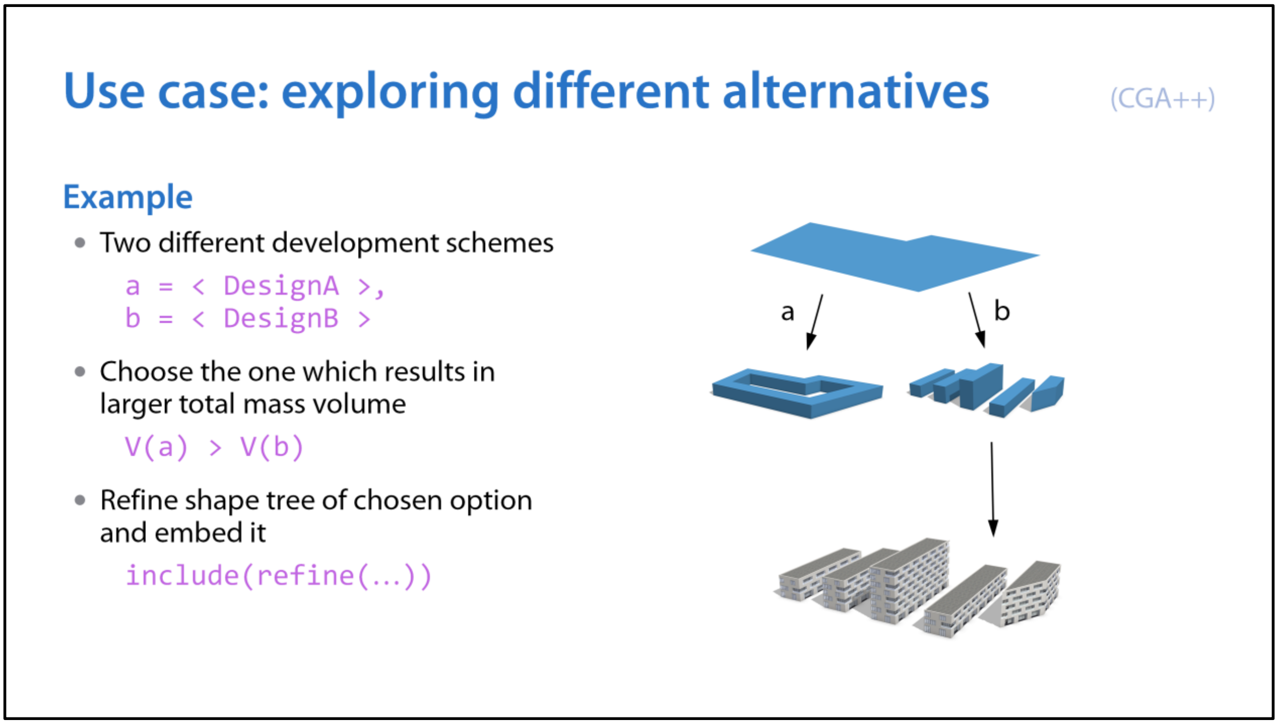
- to reason about them *e.g., to derive parameter values*
- to use them as arguments *e.g., for multi-shape operations*

Example

```
S --> minus(list(this->t('0.2, 0, '0.2)->s('0.2, '1, '0.6),
                 this->t('0.6, 0, '0.2)->s('0.2, '1, '0.6)));
```

syntactic sugar: chain operator

```
  this->t('0.2, 0, '0.2)
≡ t(this, '0.2, 0, '0.2)
```



Use case: exploring different alternatives

(CGA++)

Example

- Two different development schemes

```
a = < DesignA >,
b = < DesignB >
```

- Choose the one which results in larger total mass volume

```
V(a) > V(b)
```

- Refine shape tree of chosen option and embed it

```
include(refine(...))
```

Example is adapted from Figure 4 of the CGA++ paper.

Enhanced operations

(CGA++)

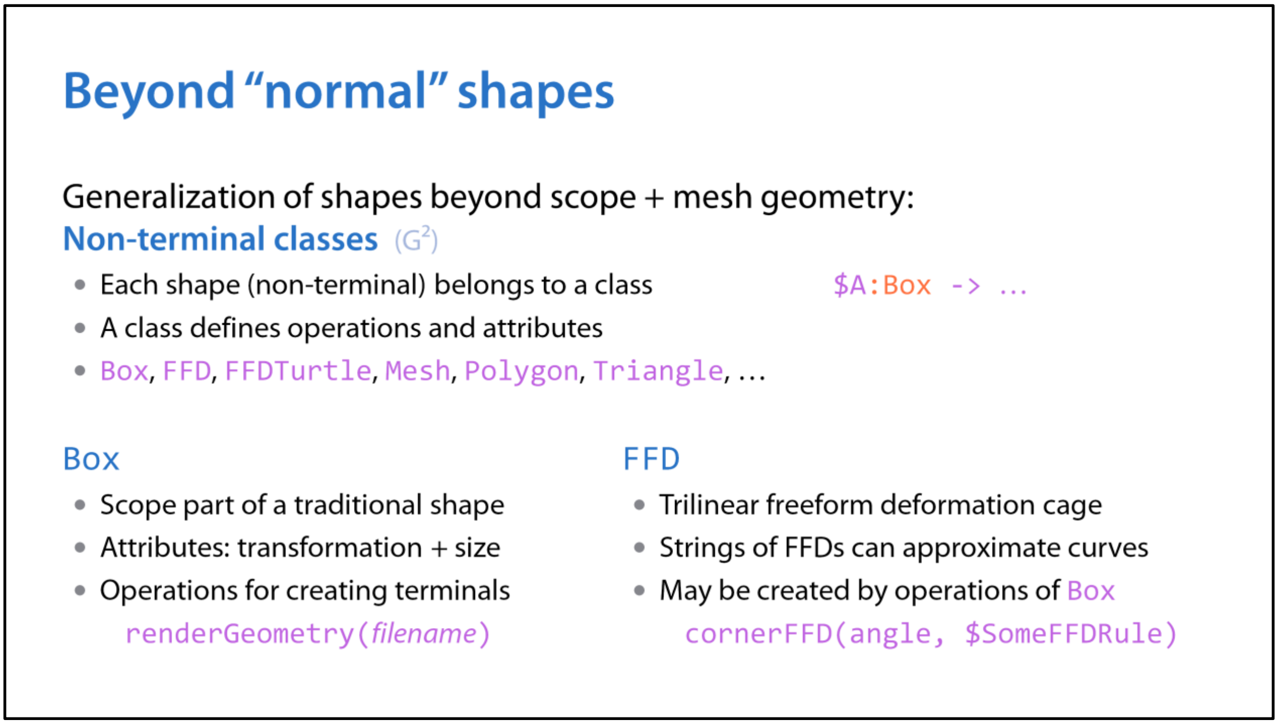
Selectors can become arbitrary predicate expressions

- Predefined selectors are exposed as (local) functions on shapes (or local variables)
- Implicit variables provide information about objects tested
- Function values are applied implicitly

Example: component split

```
comp("f") {
  top($shape)          : Roof           or simply: top
| right || front       = FrontCorner    ≡ right($shape) || front($shape)
| side && $index > 4   : Wall1
| !bottom              : Wall2
}
```

implicit variable



Beyond "normal" shapes

Generalization of shapes beyond scope + mesh geometry:

Non-terminal classes (G^2)

- Each shape (non-terminal) belongs to a class `$A:Box -> ...`
- A class defines operations and attributes
- `Box`, `FFD`, `FFDTurtle`, `Mesh`, `Polygon`, `Triangle`, …

`Box`

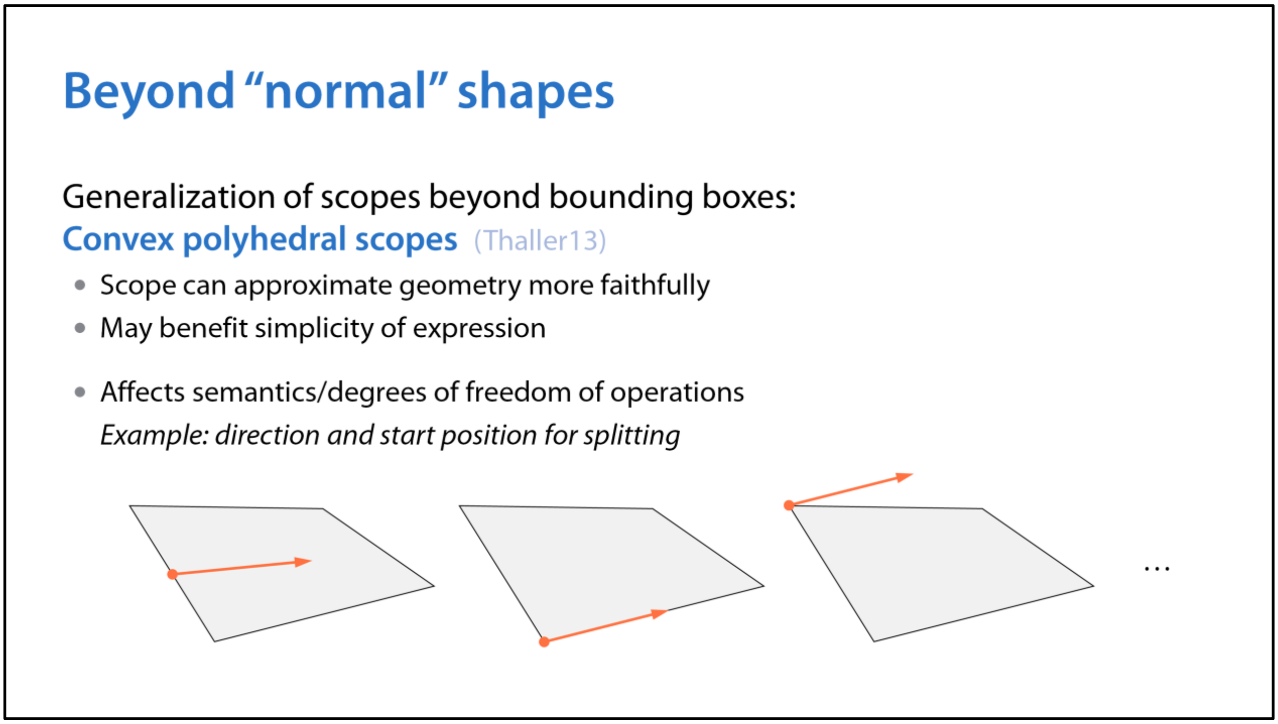
- Scope part of a traditional shape
- Attributes: transformation + size
- Operations for creating terminals

  `renderGeometry(`*filename*`)`

`FFD`

- Trilinear freeform deformation cage
- Strings of FFDs can approximate curves
- May be created by operations of `Box`

  `cornerFFD(angle, $SomeFFDRule)`



Beyond “normal” shapes

Generalization of scopes beyond bounding boxes:

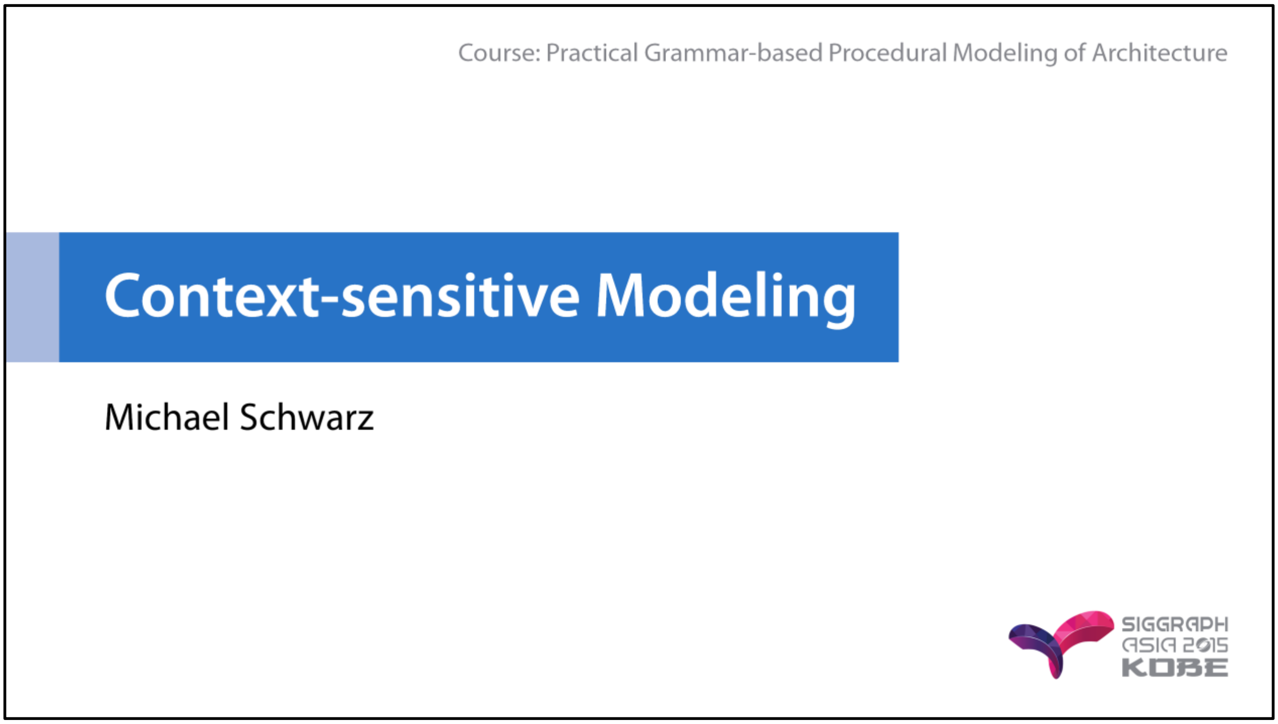
Convex polyhedral scopes (Thaller13)

- Scope can approximate geometry more faithfully
- May benefit simplicity of expression

- Affects semantics/degrees of freedom of operations

  Example: direction and start position for splitting

4 Context-sensitive modeling

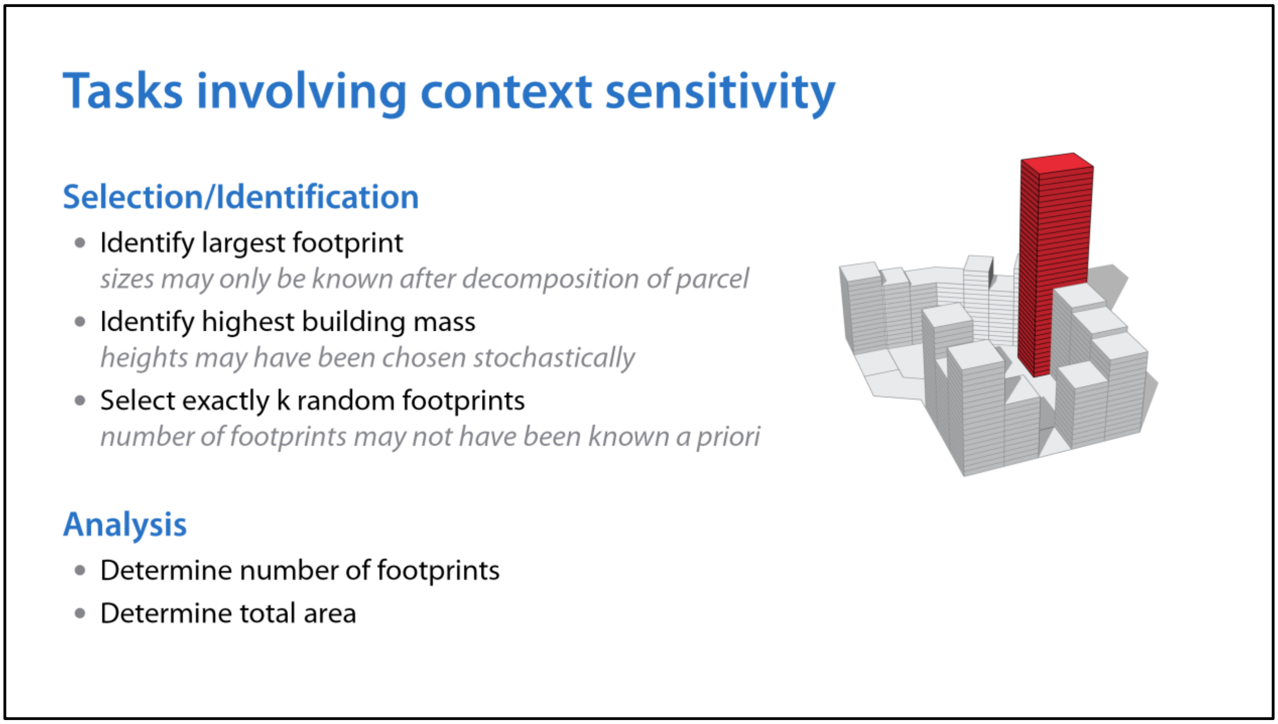


Course: Practical Grammar-based Procedural Modeling of Architecture

Context-sensitive Modeling

Michael Schwarz

SIGGRAPH ASIA 2015 KOBE



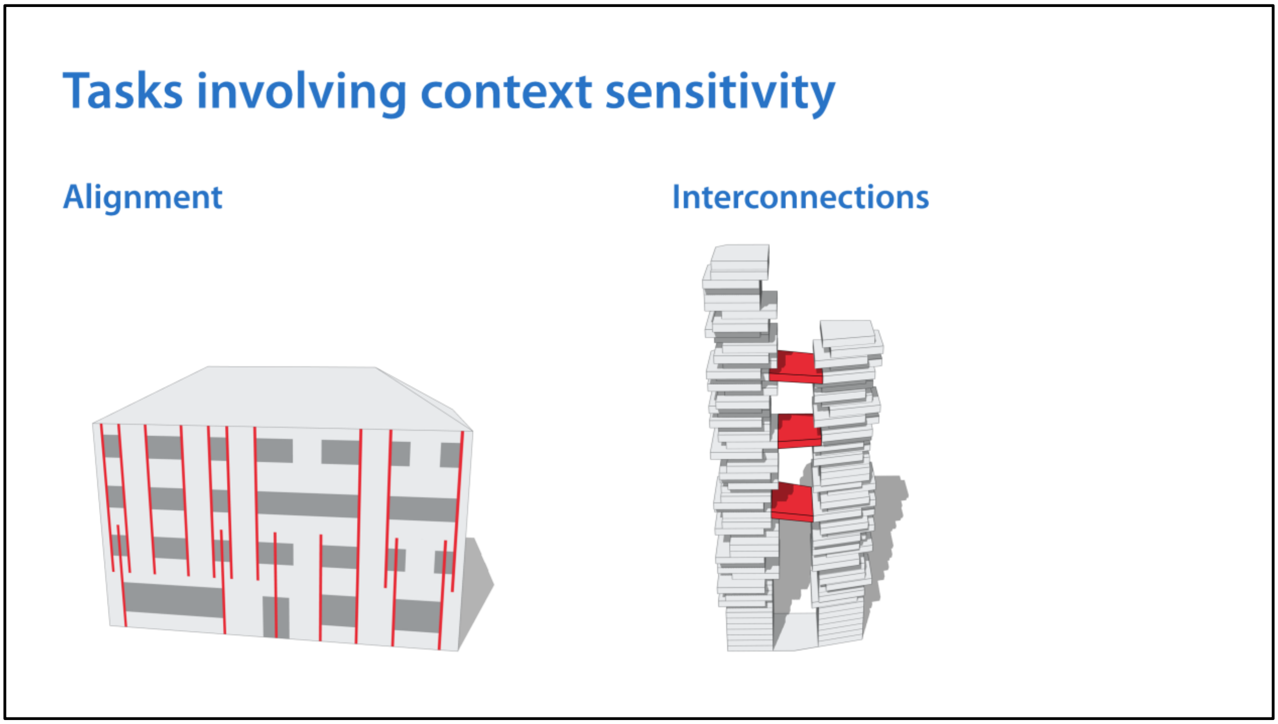
Tasks involving context sensitivity

Selection/Identification

- Identify largest footprint
  sizes may only be known after decomposition of parcel
- Identify highest building mass
  heights may have been chosen stochastically
- Select exactly k random footprints
  number of footprints may not have been known a priori

Analysis

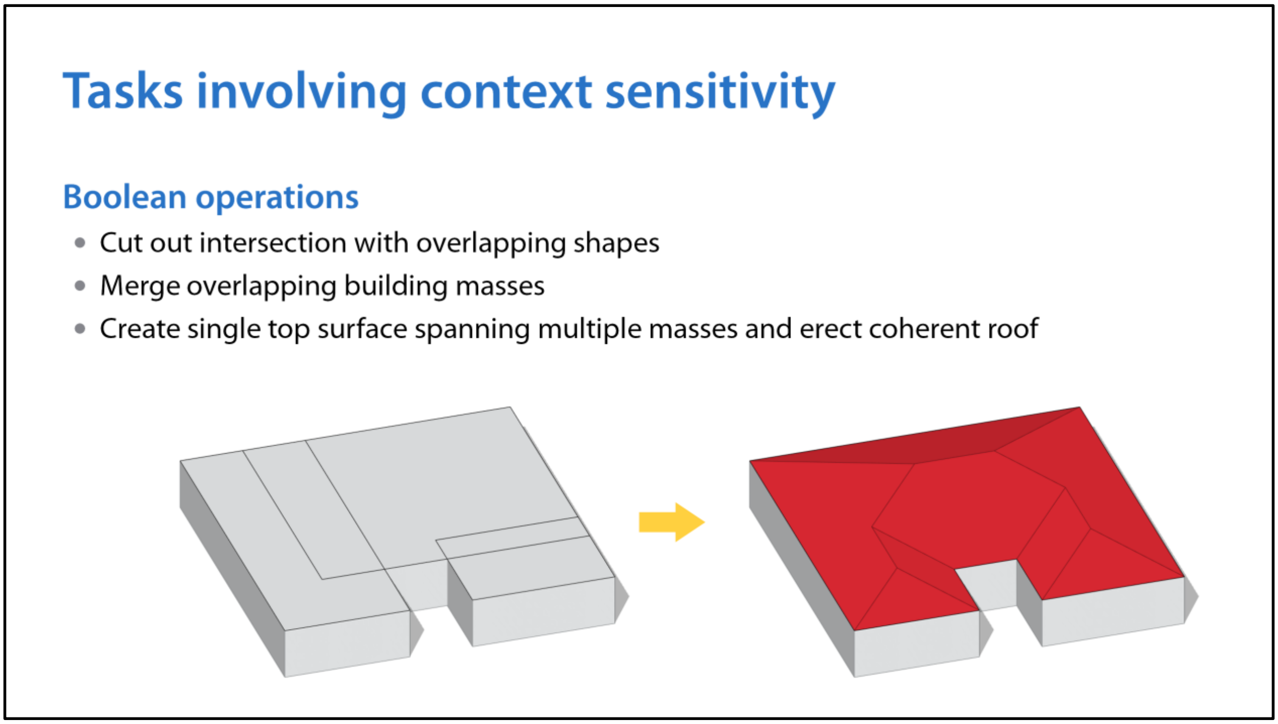
- Determine number of footprints
- Determine total area



Tasks involving context sensitivity

Alignment

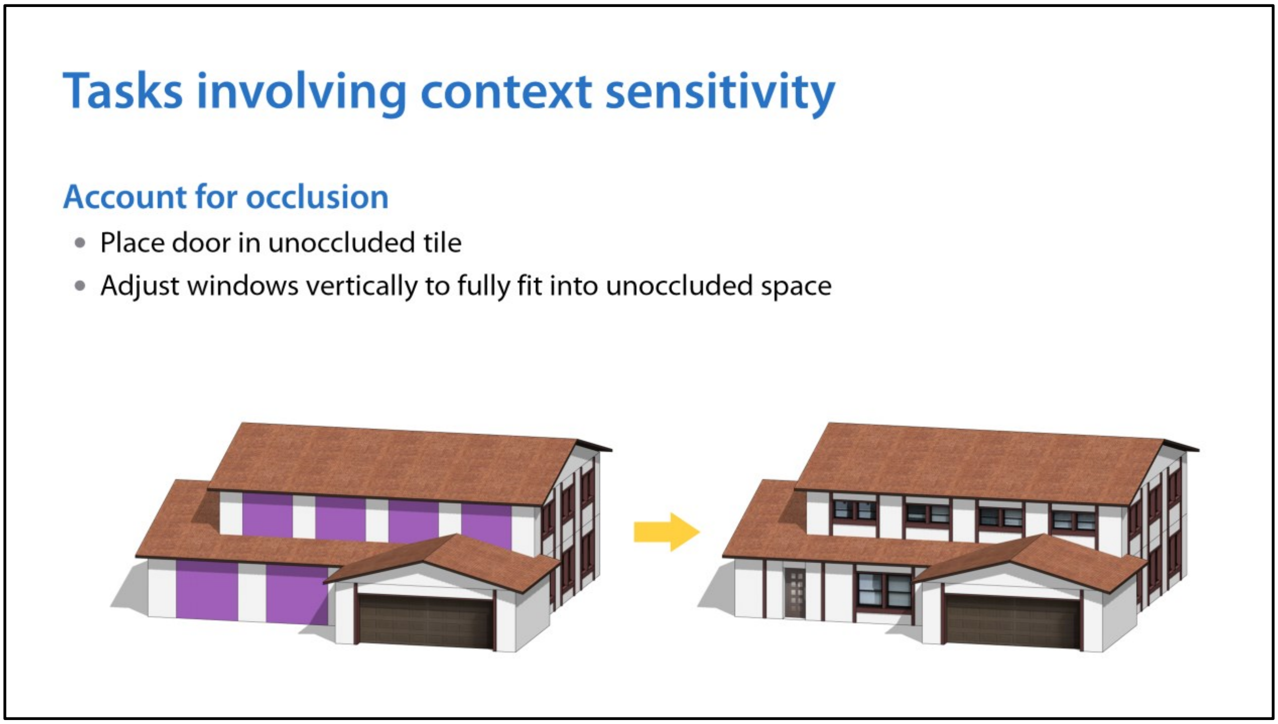
Interconnections



Tasks involving context sensitivity

Boolean operations

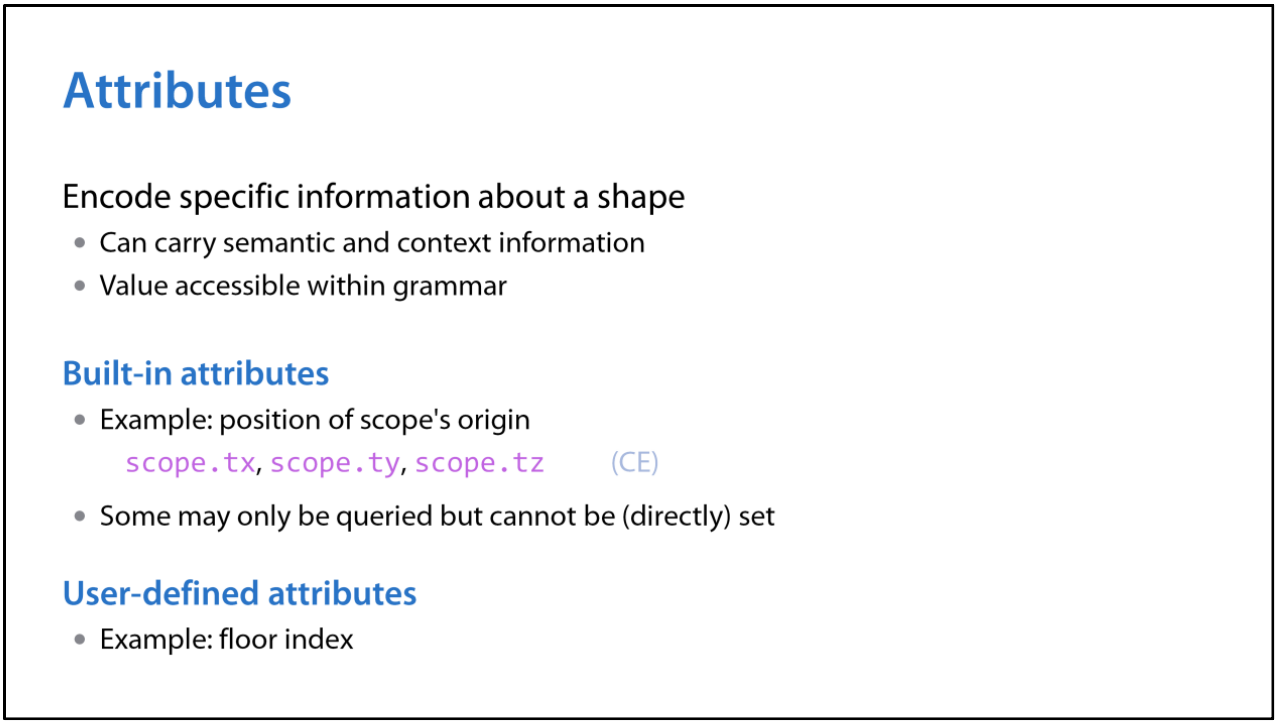
- Cut out intersection with overlapping shapes
- Merge overlapping building masses
- Create single top surface spanning multiple masses and erect coherent roof



Tasks involving context sensitivity

Account for occlusion

- Place door in unoccluded tile
- Adjust windows vertically to fully fit into unoccluded space



Attributes

Encode specific information about a shape

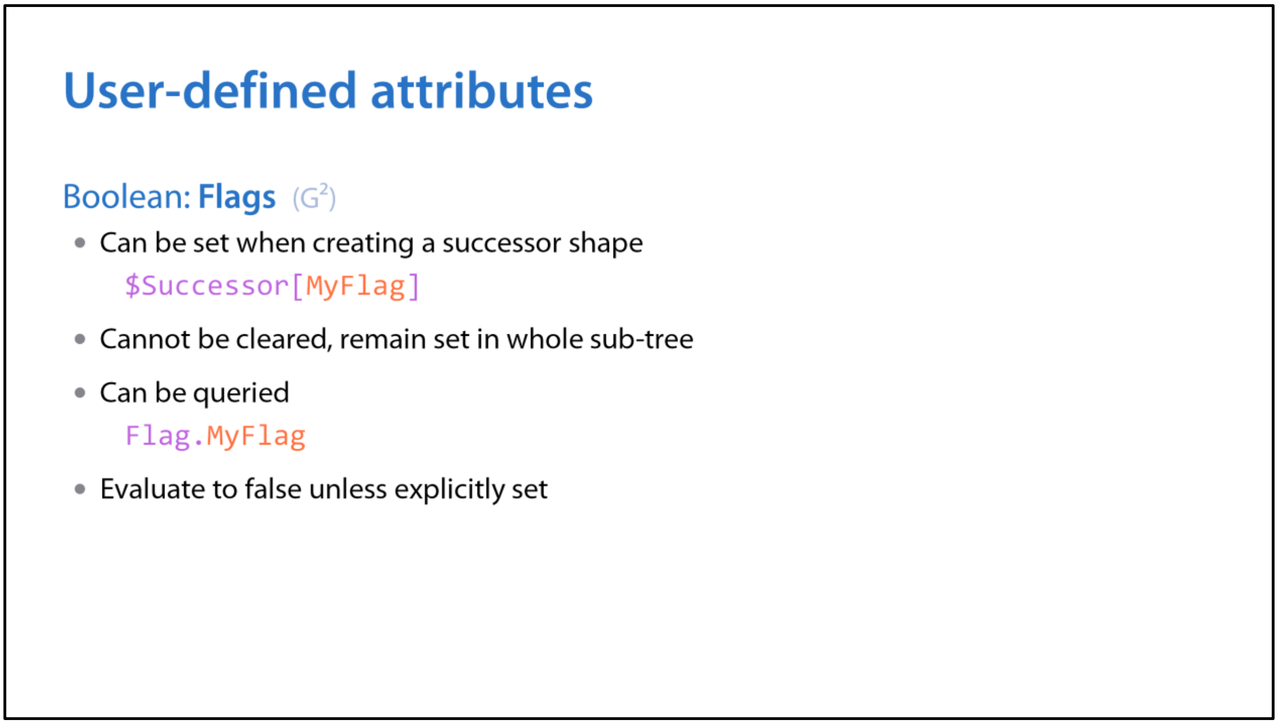
- Can carry semantic and context information
- Value accessible within grammar

Built-in attributes

- Example: position of scope's origin
  `scope.tx`, `scope.ty`, `scope.tz` (CE)
- Some may only be queried but cannot be (directly) set

User-defined attributes

- Example: floor index



User-defined attributes

Boolean: **Flags** (G²)

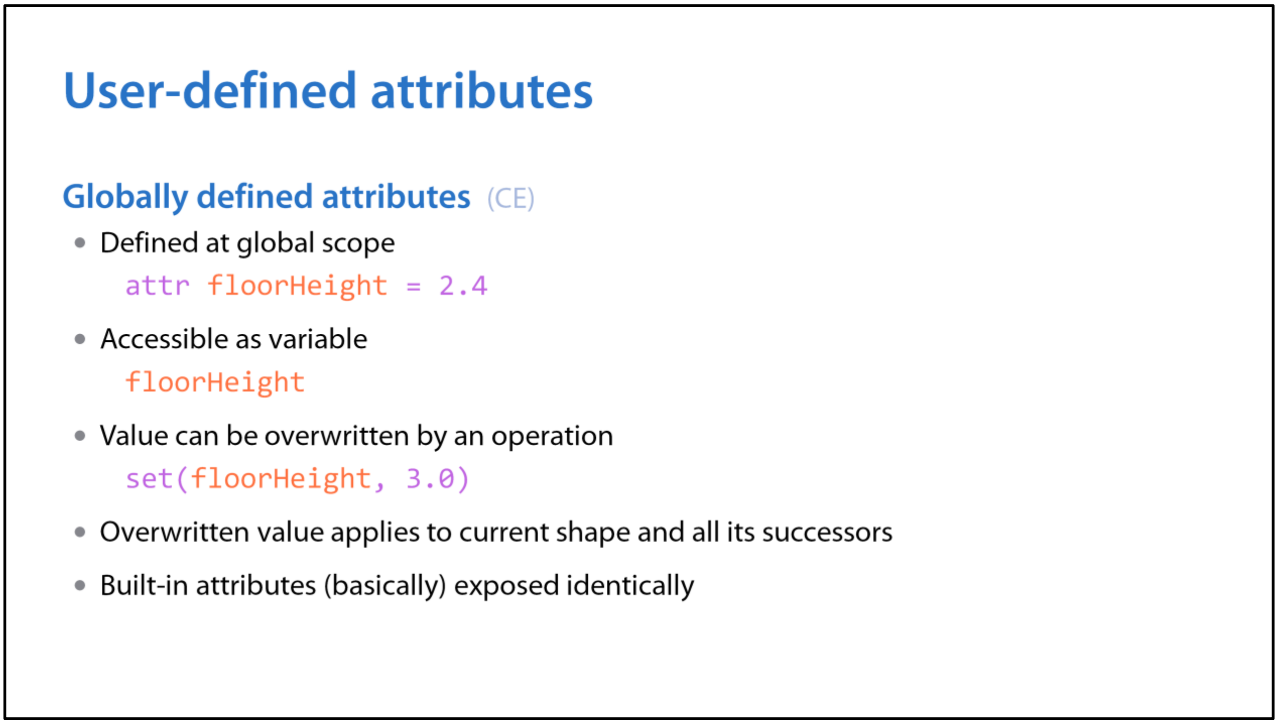
- Can be set when creating a successor shape

  `$Successor[MyFlag]`

- Cannot be cleared, remain set in whole sub-tree
- Can be queried

  `Flag.MyFlag`

- Evaluate to false unless explicitly set



User-defined attributes

Globally defined attributes (CE)

- Defined at global scope

```
attr floorHeight = 2.4
```

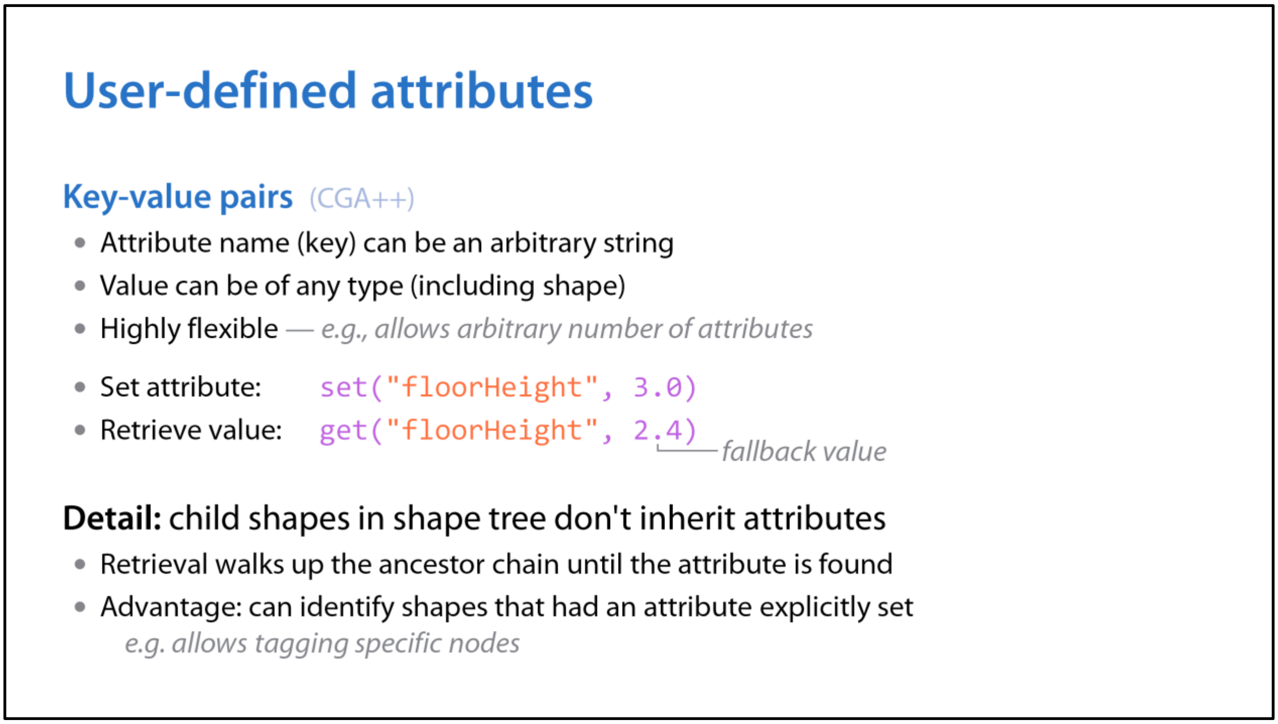
- Accessible as variable

```
floorHeight
```

- Value can be overwritten by an operation

```
set(floorHeight, 3.0)
```

- Overwritten value applies to current shape and all its successors
- Built-in attributes (basically) exposed identically



User-defined attributes

Key-value pairs (CGA++)

- Attribute name (key) can be an arbitrary string
- Value can be of any type (including shape)
- Highly flexible — *e.g., allows arbitrary number of attributes*

- Set attribute: `set("floorHeight", 3.0)`
- Retrieve value: `get("floorHeight", 2.4)` *fallback value*

Detail: child shapes in shape tree don't inherit attributes

- Retrieval walks up the ancestor chain until the attribute is found
- Advantage: can identify shapes that had an attribute explicitly set
  e.g. allows tagging specific nodes

Context information from operations

Encoding options

- Built-in attribute
  set for each successor shape
  `split.index` (CE)
- Special variable
  visible to operation's arguments
  `Operator.index` (G^2)

Examples

- Split operation (CE)

`split.index`,	*index of successors*
`split.total`	*number of successors*

- Component split (CE)

`comp.sel`,	*selector*
`comp.index`, `comp.total`	

- Extrusion (G^2)

`Operator`

`.index`	*index of edge*
`.la`	*edge's left outer angle*
`.ra`	*edge's right outer angle*

Involvement of other shapes

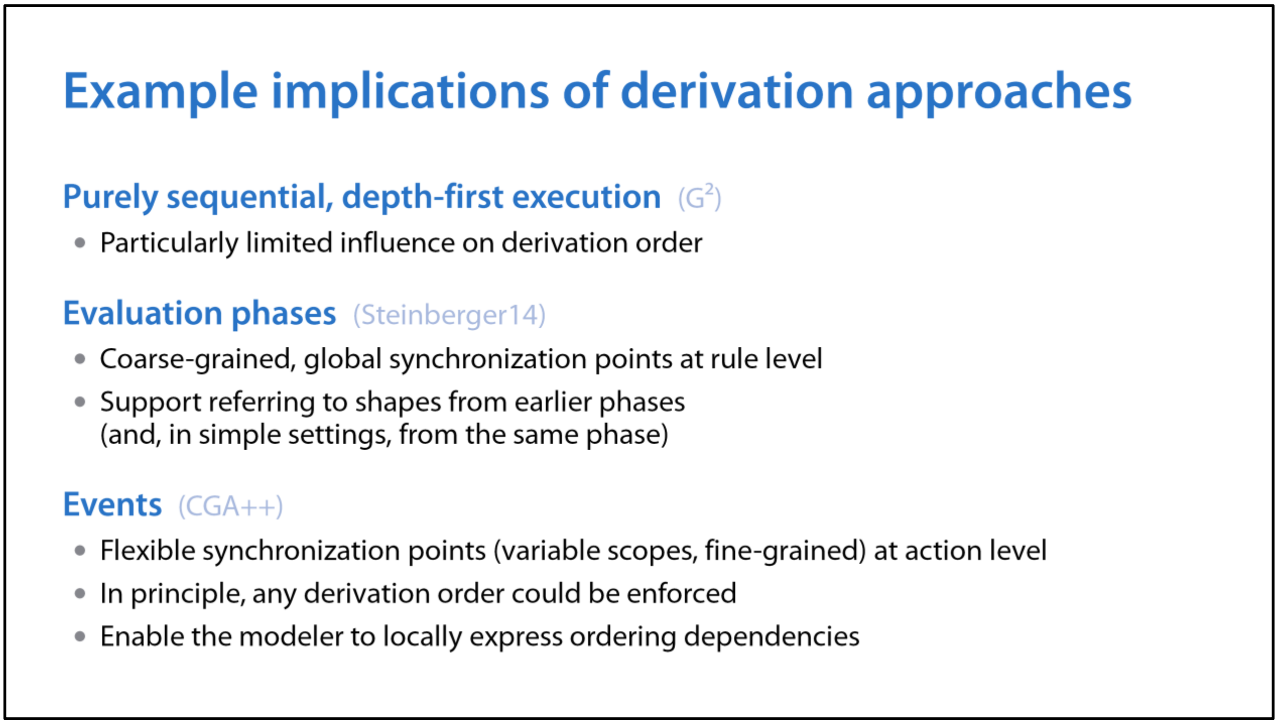
Context-sensitive modeling often requires referring to other shapes

Prerequisite: these shapes must exist when establishing the context

Important role: **derivation process/order** and available **means to guide it**

They influence

- what contexts are possible
- whether the set of involved shapes is well defined and deterministic/reproducible
- the actual effort required to ensure that shapes exist



Example implications of derivation approaches

Purely sequential, depth-first execution (G^2)

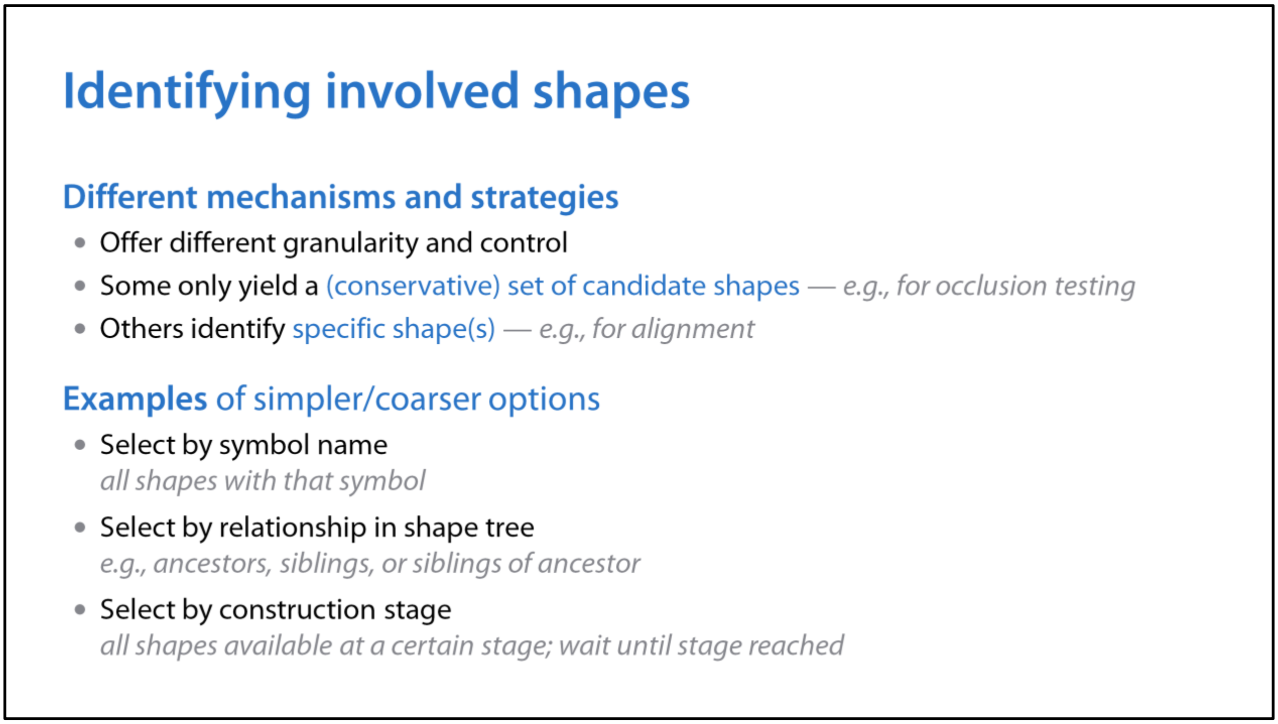
- Particularly limited influence on derivation order

Evaluation phases (Steinberger14)

- Coarse-grained, global synchronization points at rule level
- Support referring to shapes from earlier phases
  (and, in simple settings, from the same phase)

Events (CGA++)

- Flexible synchronization points (variable scopes, fine-grained) at action level
- In principle, any derivation order could be enforced
- Enable the modeler to locally express ordering dependencies



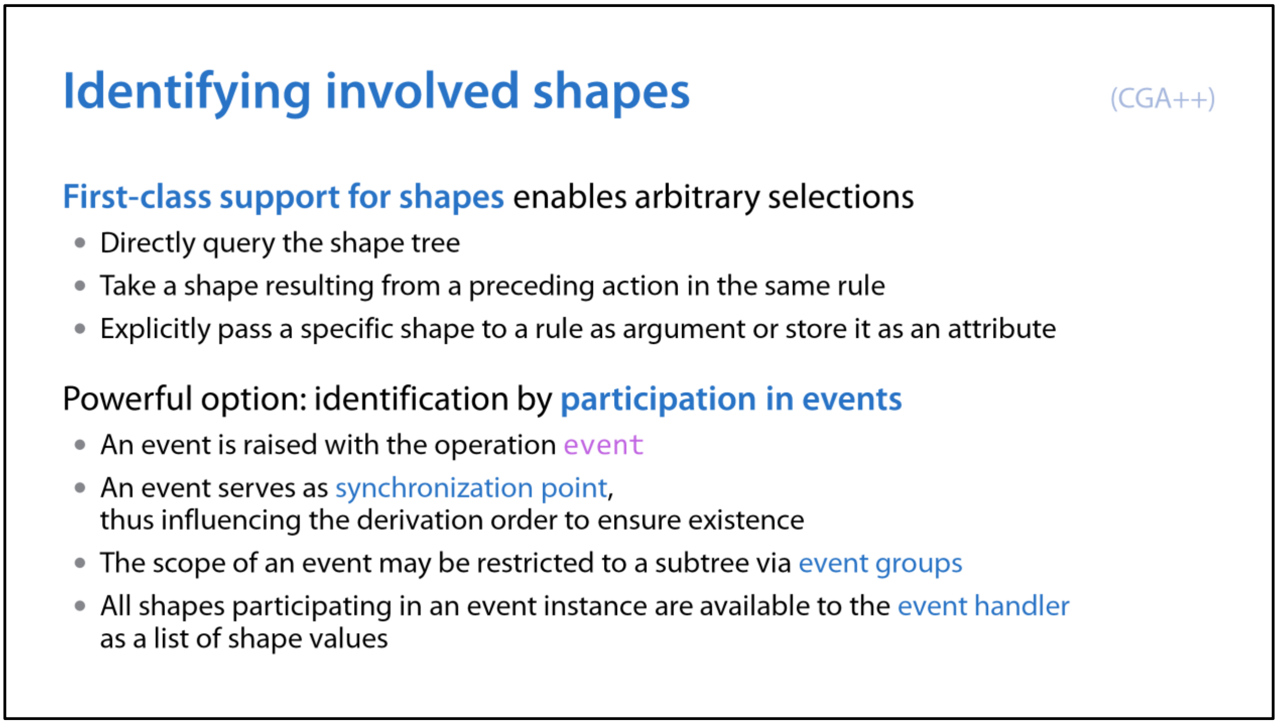
Identifying involved shapes

Different mechanisms and strategies

- Offer different granularity and control
- Some only yield a (conservative) set of candidate shapes — *e.g., for occlusion testing*
- Others identify specific shape(s) — *e.g., for alignment*

Examples of simpler/coarser options

- Select by symbol name
  all shapes with that symbol
- Select by relationship in shape tree
  e.g., ancestors, siblings, or siblings of ancestor
- Select by construction stage
  all shapes available at a certain stage; wait until stage reached



Identifying involved shapes

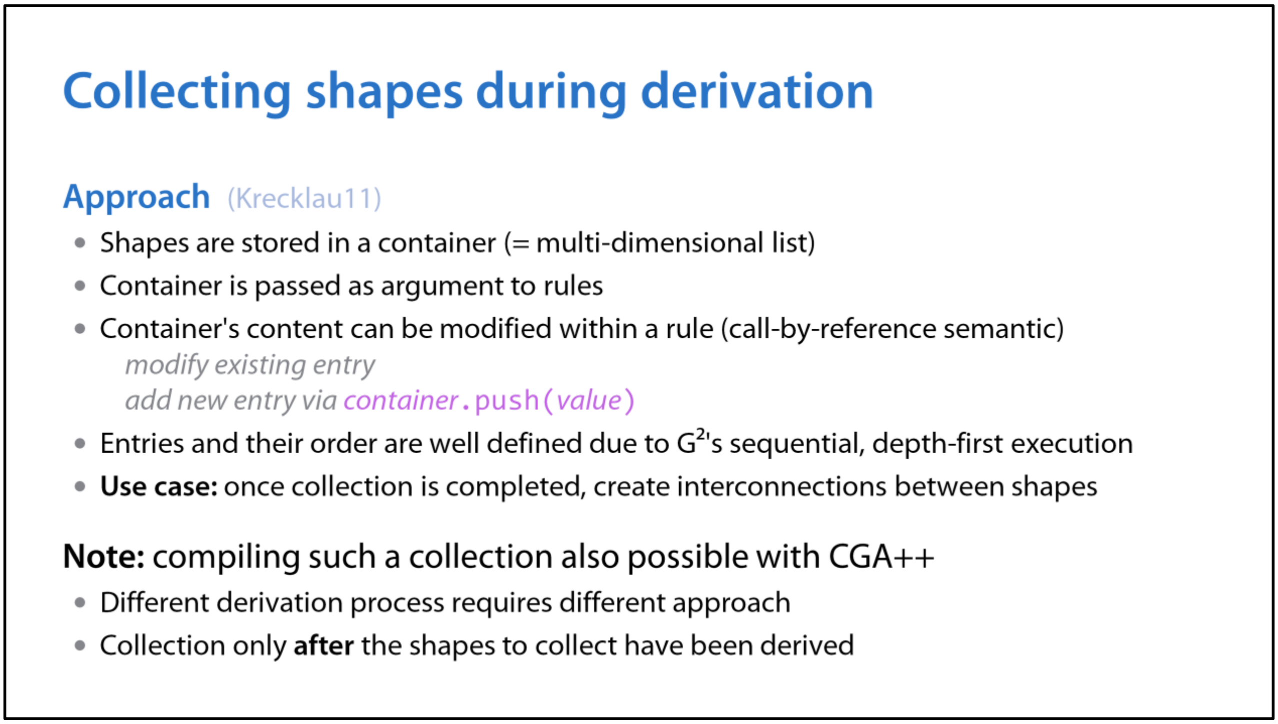
(CGA++)

First-class support for shapes enables arbitrary selections

- Directly query the shape tree
- Take a shape resulting from a preceding action in the same rule
- Explicitly pass a specific shape to a rule as argument or store it as an attribute

Powerful option: identification by **participation in events**

- An event is raised with the operation `event`
- An event serves as synchronization point,
  thus influencing the derivation order to ensure existence
- The scope of an event may be restricted to a subtree via event groups
- All shapes participating in an event instance are available to the event handler
  as a list of shape values



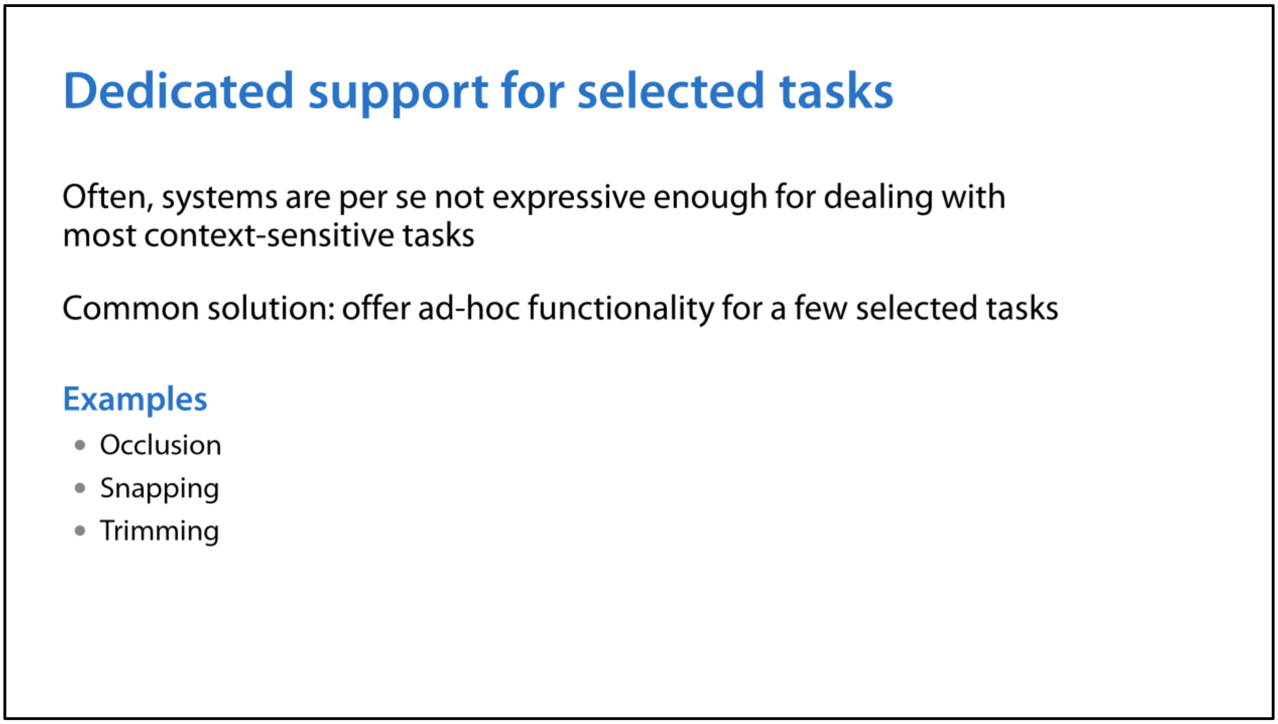
Collecting shapes during derivation

Approach (Krecklau11)

- Shapes are stored in a container (= multi-dimensional list)
- Container is passed as argument to rules
- Container's content can be modified within a rule (call-by-reference semantic)
  - *modify existing entry*
  - *add new entry via* `container.push(value)`
- Entries and their order are well defined due to G^2's sequential, depth-first execution
- **Use case:** once collection is completed, create interconnections between shapes

Note: compiling such a collection also possible with CGA++

- Different derivation process requires different approach
- Collection only **after** the shapes to collect have been derived



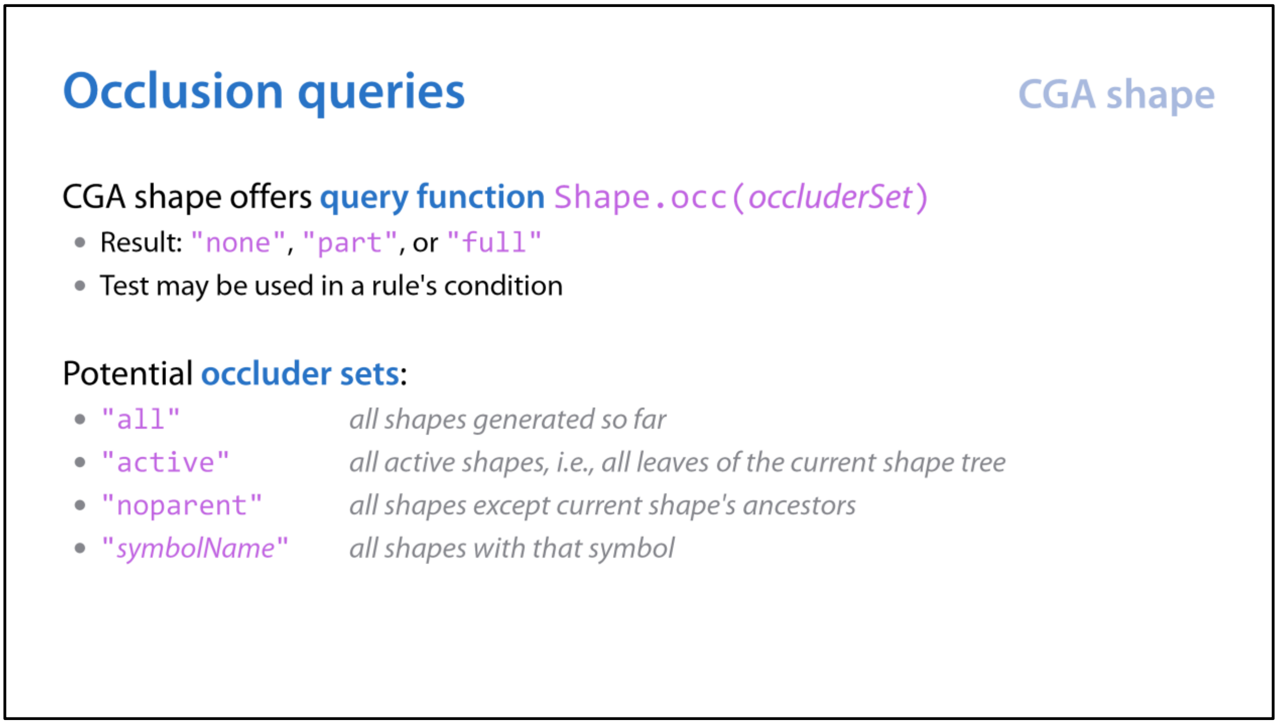
Dedicated support for selected tasks

Often, systems are per se not expressive enough for dealing with most context-sensitive tasks

Common solution: offer ad-hoc functionality for a few selected tasks

Examples

- Occlusion
- Snapping
- Trimming



Occlusion queries

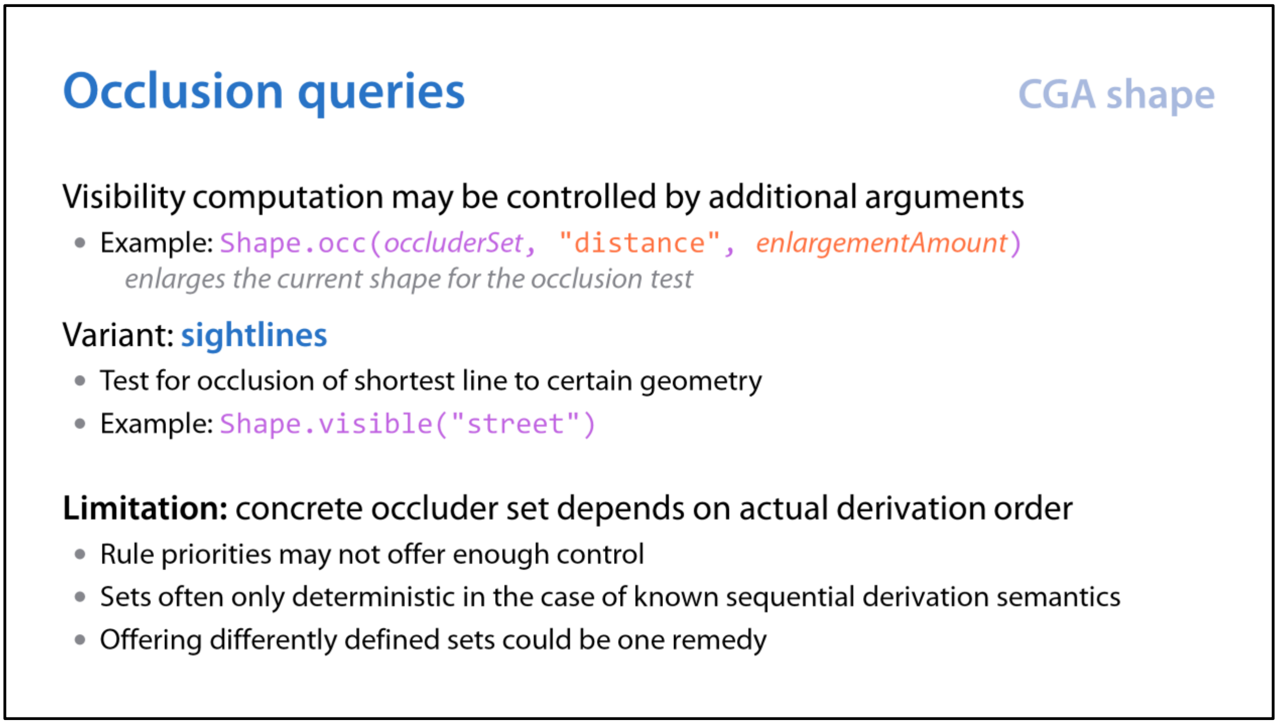
CGA shape

CGA shape offers **query function** `Shape.occ(`*occluderSet*`)`

- Result: `"none"`, `"part"`, or `"full"`
- Test may be used in a rule's condition

Potential **occluder sets**:

`"all"`	*all shapes generated so far*
`"active"`	*all active shapes, i.e., all leaves of the current shape tree*
`"noparent"`	*all shapes except current shape's ancestors*
"*symbolName*"	*all shapes with that symbol*



Occlusion queries

CGA shape

Visibility computation may be controlled by additional arguments

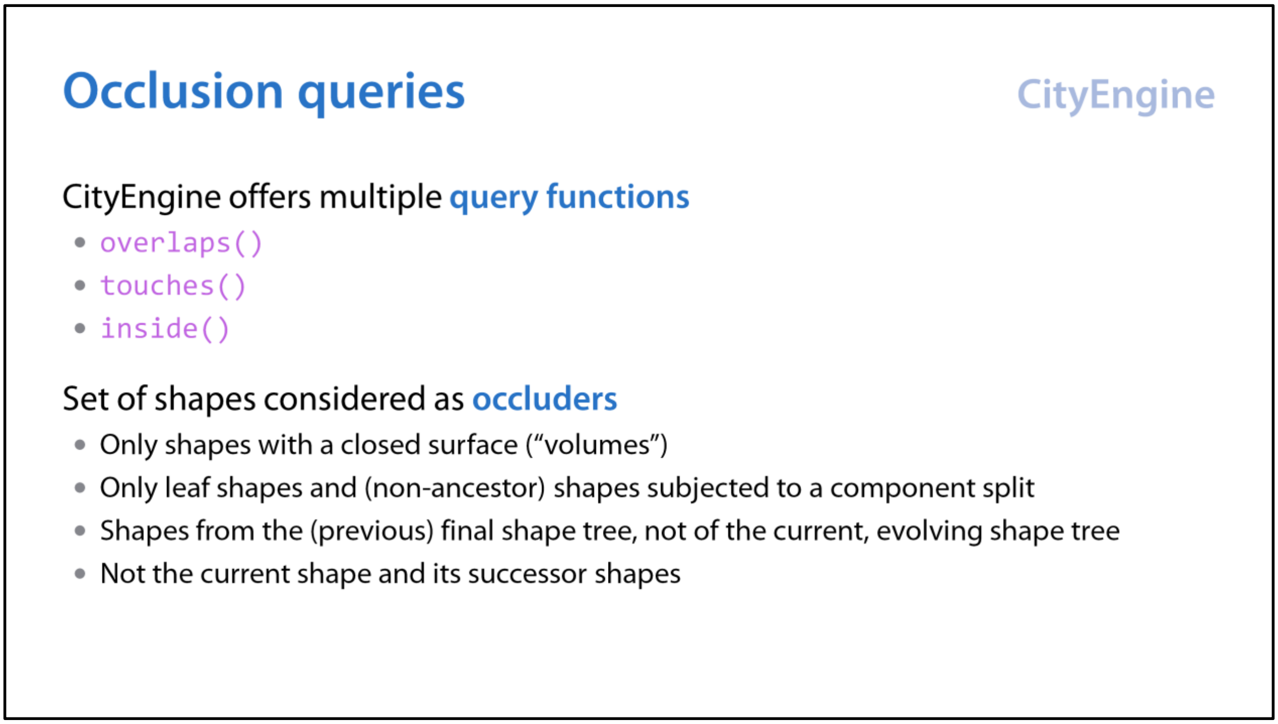
- Example: `Shape.occ(occluderSet, "distance", enlargementAmount)`
  enlarges the current shape for the occlusion test

Variant: **sightlines**

- Test for occlusion of shortest line to certain geometry
- Example: `Shape.visible("street")`

Limitation: concrete occluder set depends on actual derivation order

- Rule priorities may not offer enough control
- Sets often only deterministic in the case of known sequential derivation semantics
- Offering differently defined sets could be one remedy



Occlusion queries

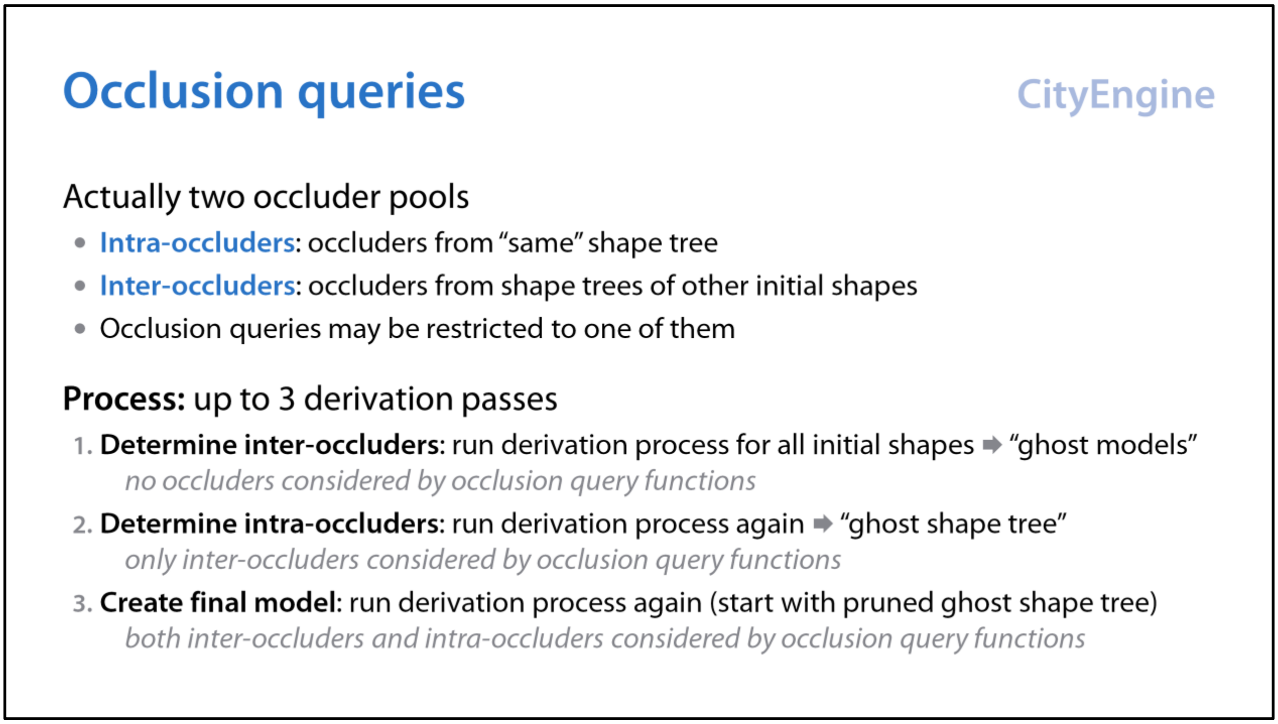
CityEngine

CityEngine offers multiple **query functions**

- `overlaps()`
- `touches()`
- `inside()`

Set of shapes considered as **occluders**

- Only shapes with a closed surface (“volumes”)
- Only leaf shapes and (non-ancestor) shapes subjected to a component split
- Shapes from the (previous) final shape tree, not of the current, evolving shape tree
- Not the current shape and its successor shapes



Occlusion queries

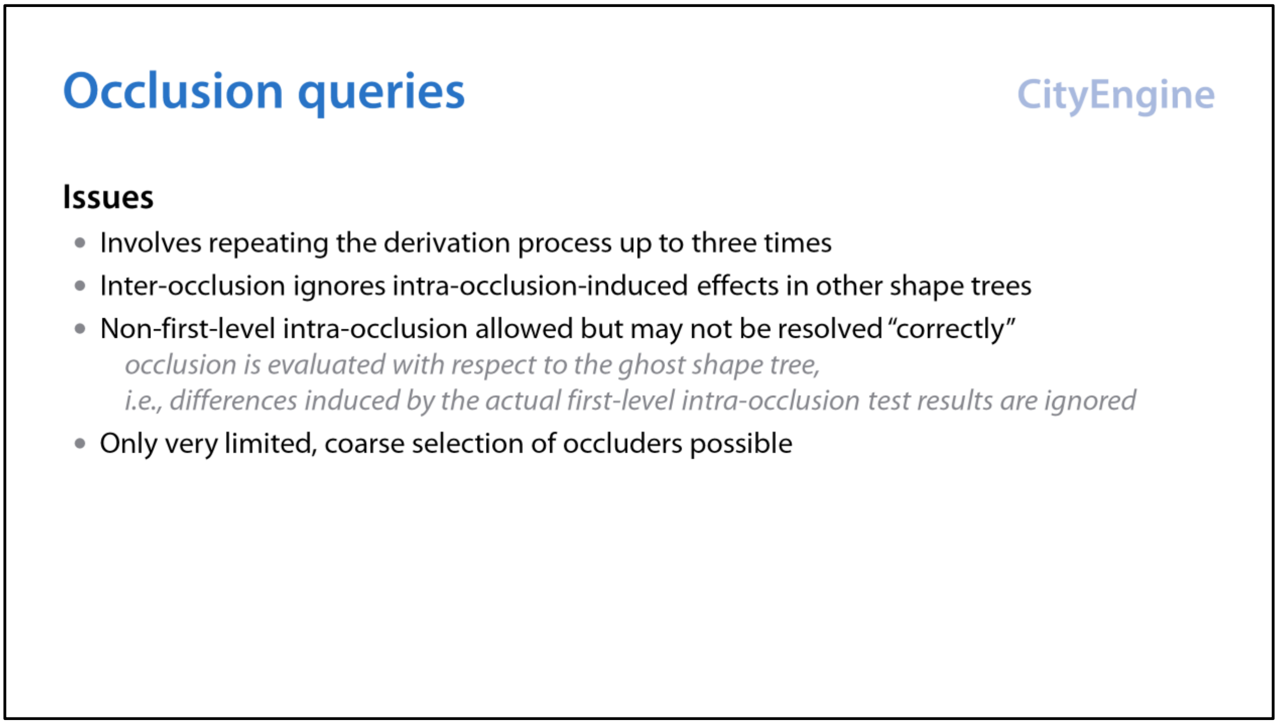
CityEngine

Actually two occluder pools

- **Intra-occluders**: occluders from “same” shape tree
- **Inter-occluders**: occluders from shape trees of other initial shapes
- Occlusion queries may be restricted to one of them

Process: up to 3 derivation passes

1. **Determine inter-occluders:** run derivation process for all initial shapes ➡ “ghost models”
   no occluders considered by occlusion query functions
2. **Determine intra-occluders:** run derivation process again ➡ “ghost shape tree”
   only inter-occluders considered by occlusion query functions
3. **Create final model:** run derivation process again (start with pruned ghost shape tree)
   both inter-occluders and intra-occluders considered by occlusion query functions

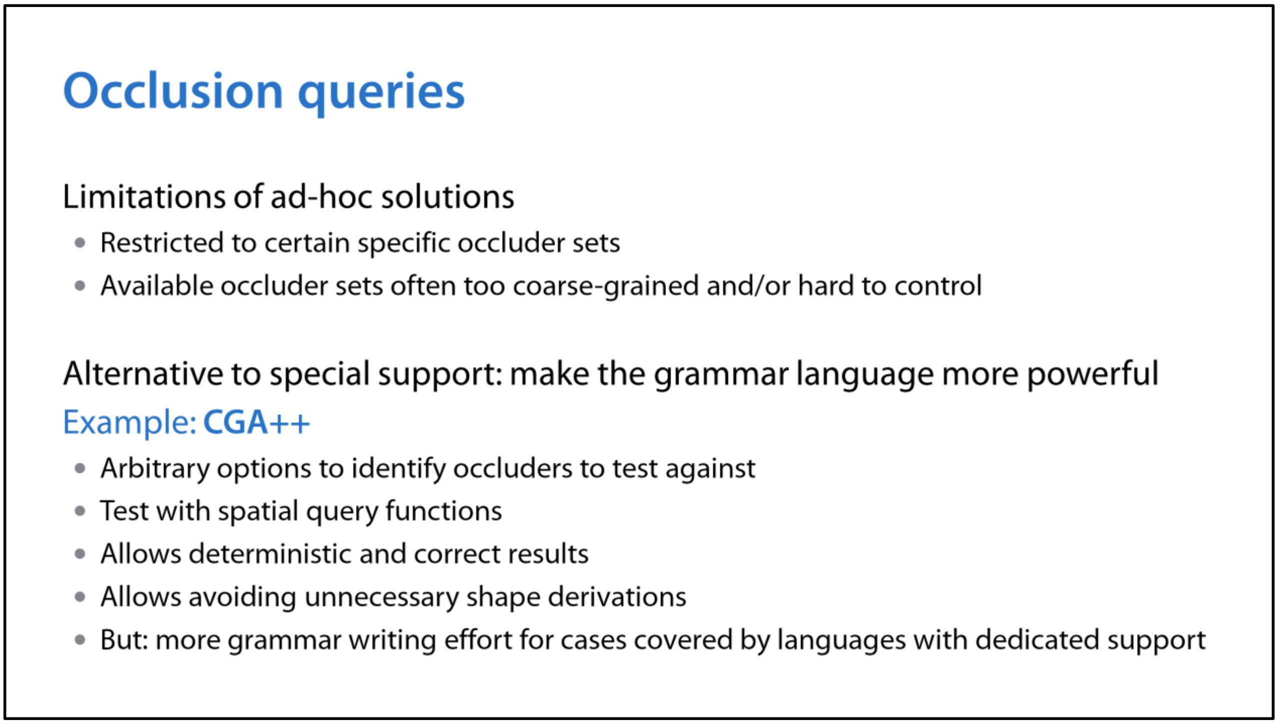


Occlusion queries

CityEngine

Issues

- Involves repeating the derivation process up to three times
- Inter-occlusion ignores intra-occlusion-induced effects in other shape trees
- Non-first-level intra-occlusion allowed but may not be resolved "correctly"
  occlusion is evaluated with respect to the ghost shape tree,
  i.e., differences induced by the actual first-level intra-occlusion test results are ignored
- Only very limited, coarse selection of occluders possible



Occlusion queries

Limitations of ad-hoc solutions

- Restricted to certain specific occluder sets
- Available occluder sets often too coarse-grained and/or hard to control

Alternative to special support: make the grammar language more powerful

Example: **CGA++**

- Arbitrary options to identify occluders to test against
- Test with spatial query functions
- Allows deterministic and correct results
- Allows avoiding unnecessary shape derivations
- But: more grammar writing effort for cases covered by languages with dedicated support

Occlusion

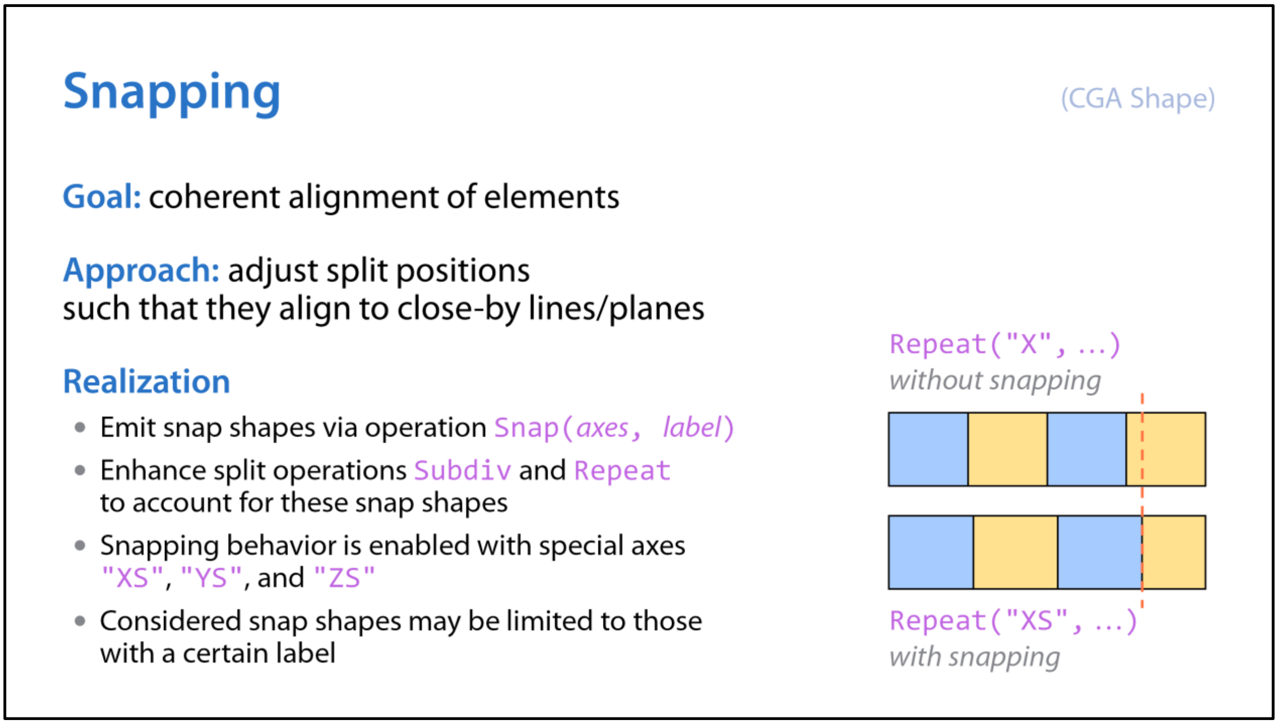
Occlusion queries only tell the degree of occlusion

- Actual occluders remain unknown
- Limits possible reactions

Unless the language is powerful enough:
enabling a certain more advanced reaction requires
an according special operation

Example: remove all occluded parts of the current shape

- New split operation `unoccludedParts` (Schwarz14)
- Combines occlusion test with Boolean operation



Snapping

(CGA Shape)

Goal: coherent alignment of elements

Approach: adjust split positions
such that they align to close-by lines/planes

Realization

- Emit snap shapes via operation `Snap(`*axes*`,` *label*`)`
- Enhance split operations `Subdiv` and `Repeat` to account for these snap shapes
- Snapping behavior is enabled with special axes `"XS"`, `"YS"`, and `"ZS"`
- Considered snap shapes may be limited to those with a certain label

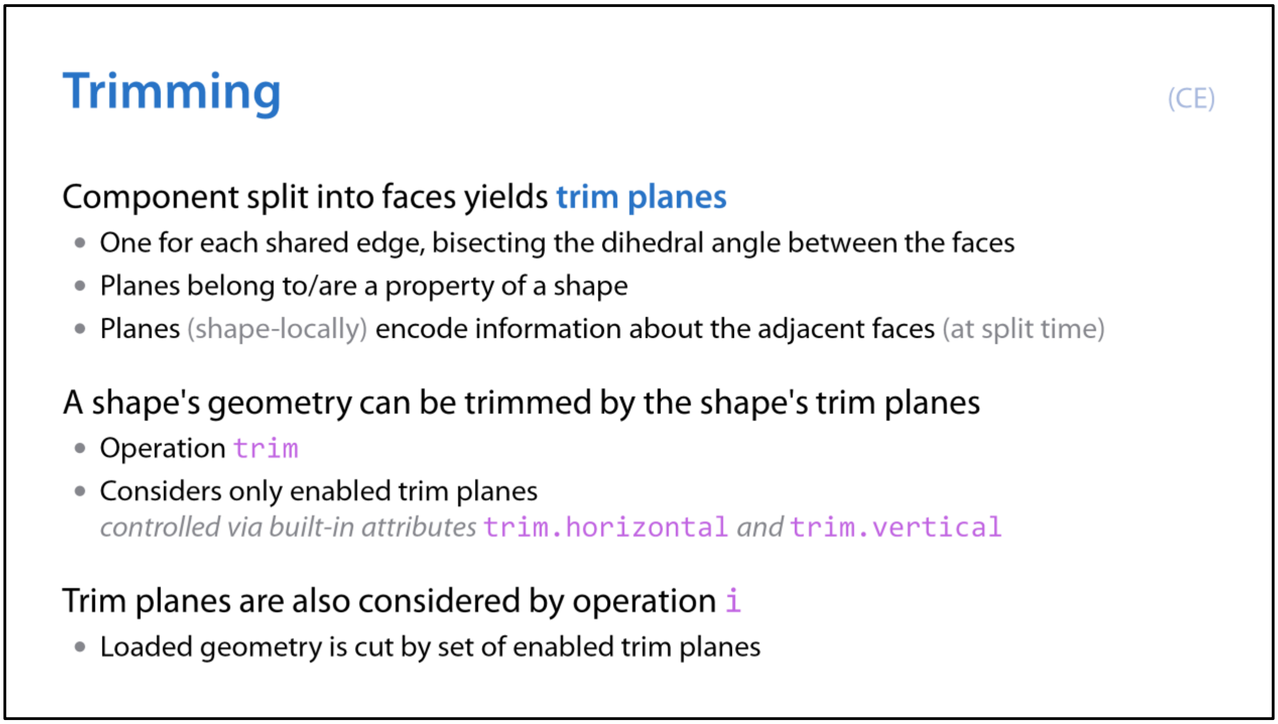
`Repeat("X", ...)`
without snapping

`Repeat("XS", ...)`
with snapping

Snapping

Related approach: **avoidance volumes** (Thaller13)

- Adjust split positions such that overlap with certain shapes is avoided
- Allowed movement of split position is bounded by maximum distance
- Enhanced split operations take a list of shapes to avoid
- **Application:** avoid placing interior walls behind windows



Trimming

(CE)

Component split into faces yields **trim planes**

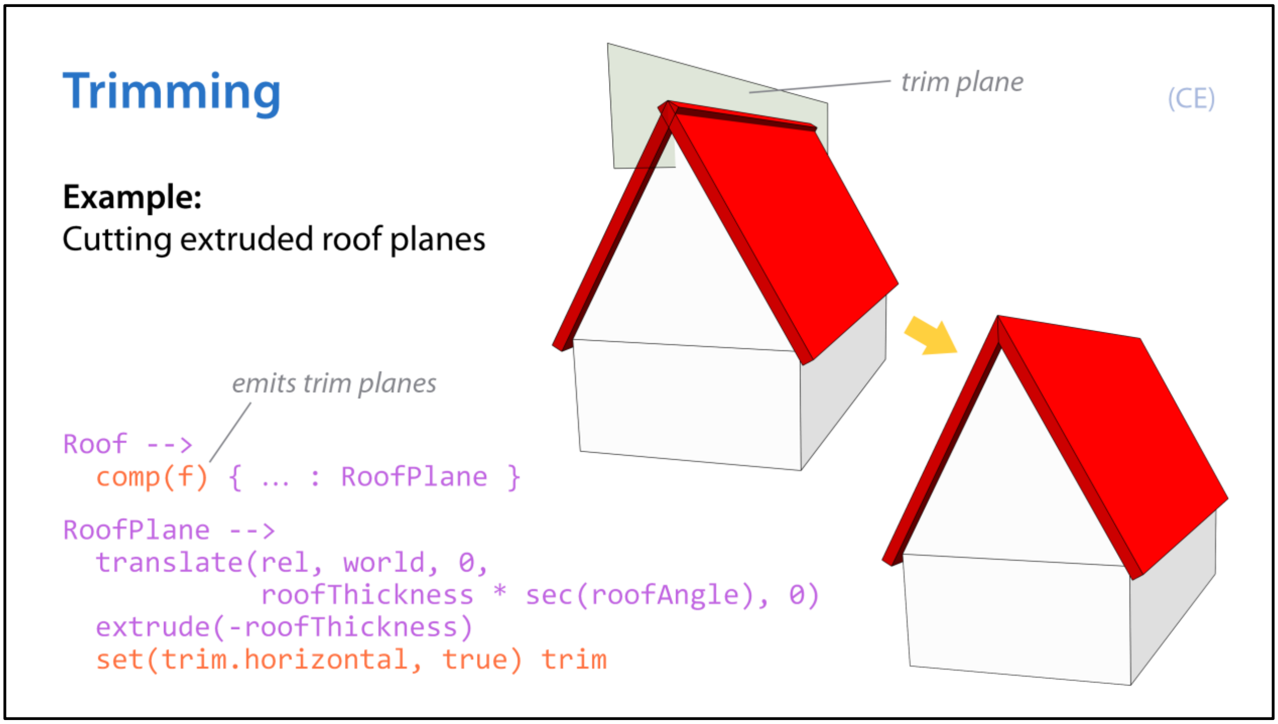
- One for each shared edge, bisecting the dihedral angle between the faces
- Planes belong to/are a property of a shape
- Planes (shape-locally) encode information about the adjacent faces (at split time)

A shape's geometry can be trimmed by the shape's trim planes

- Operation `trim`
- Considers only enabled trim planes
  controlled via built-in attributes `trim.horizontal` *and* `trim.vertical`

Trim planes are also considered by operation `i`

- Loaded geometry is cut by set of enabled trim planes



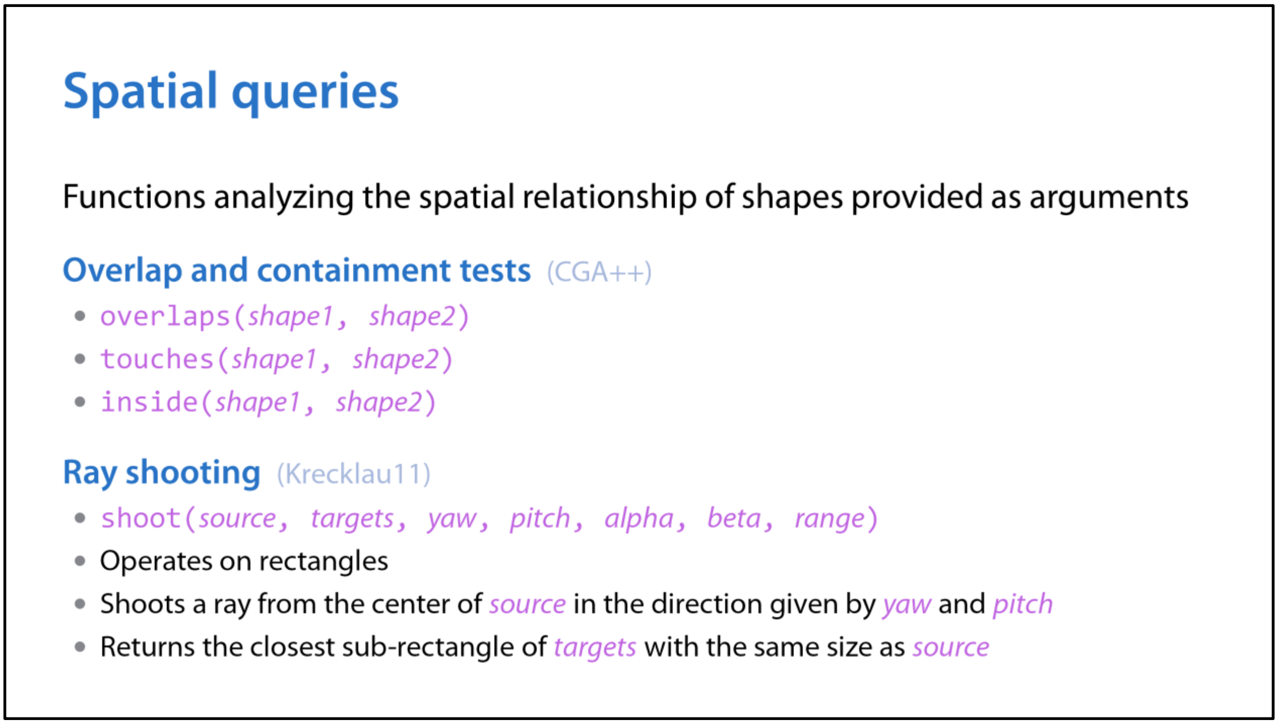
Trimming

(CE)

Example:
Cutting extruded roof planes

```
Roof -->
  comp(f) { … : RoofPlane }

RoofPlane -->
  translate(rel, world, 0,
            roofThickness * sec(roofAngle), 0)
  extrude(-roofThickness)
  set(trim.horizontal, true) trim
```

```
...
roofGable(roofAngle,
          overhangX + roofThickness * tan(roofAngle),
          overhangY)
Roof
...

square(x) = x*x
sec(x) = sqrt(1 + square(tan(x)))
```



Spatial queries

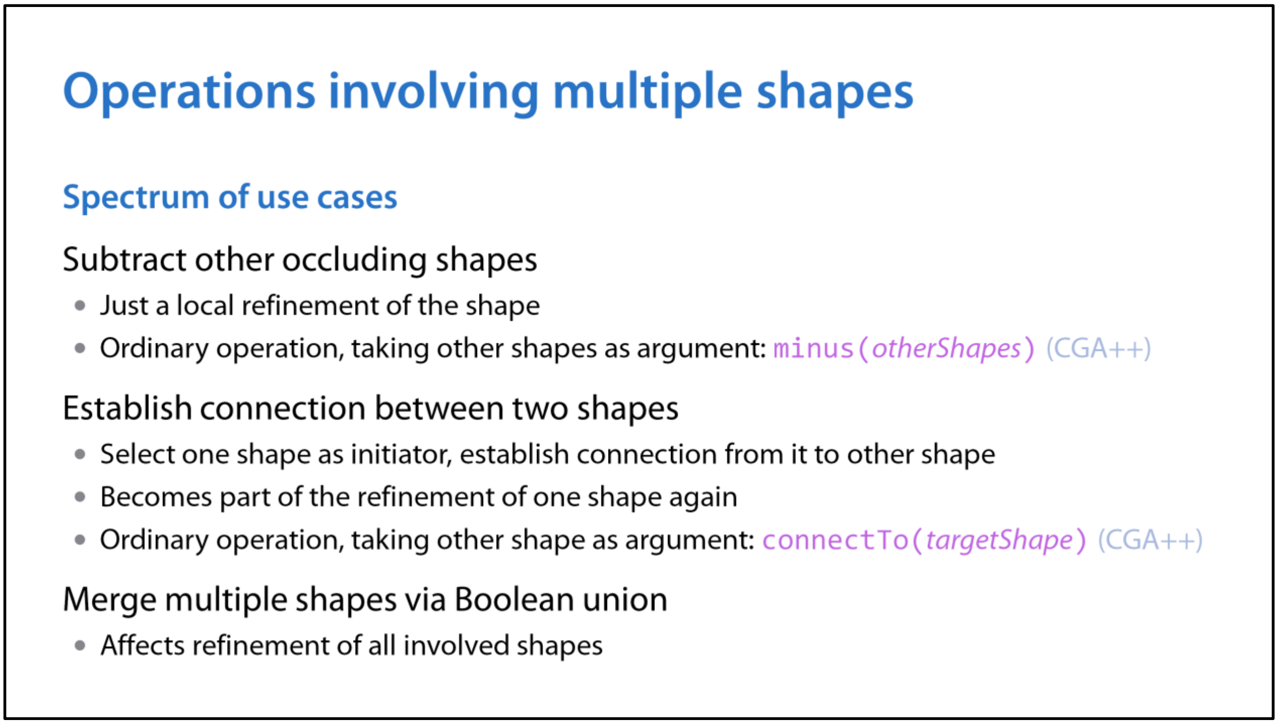
Functions analyzing the spatial relationship of shapes provided as arguments

Overlap and containment tests (CGA++)

- `overlaps(`*shape1*`, `*shape2*`)`
- `touches(`*shape1*`, `*shape2*`)`
- `inside(`*shape1*`, `*shape2*`)`

Ray shooting (Krecklau11)

- `shoot(`*source*`, `*targets*`, `*yaw*`, `*pitch*`, `*alpha*`, `*beta*`, `*range*`)`
- Operates on rectangles
- Shoots a ray from the center of *source* in the direction given by *yaw* and *pitch*
- Returns the closest sub-rectangle of *targets* with the same size as *source*



Operations involving multiple shapes

Spectrum of use cases

Subtract other occluding shapes

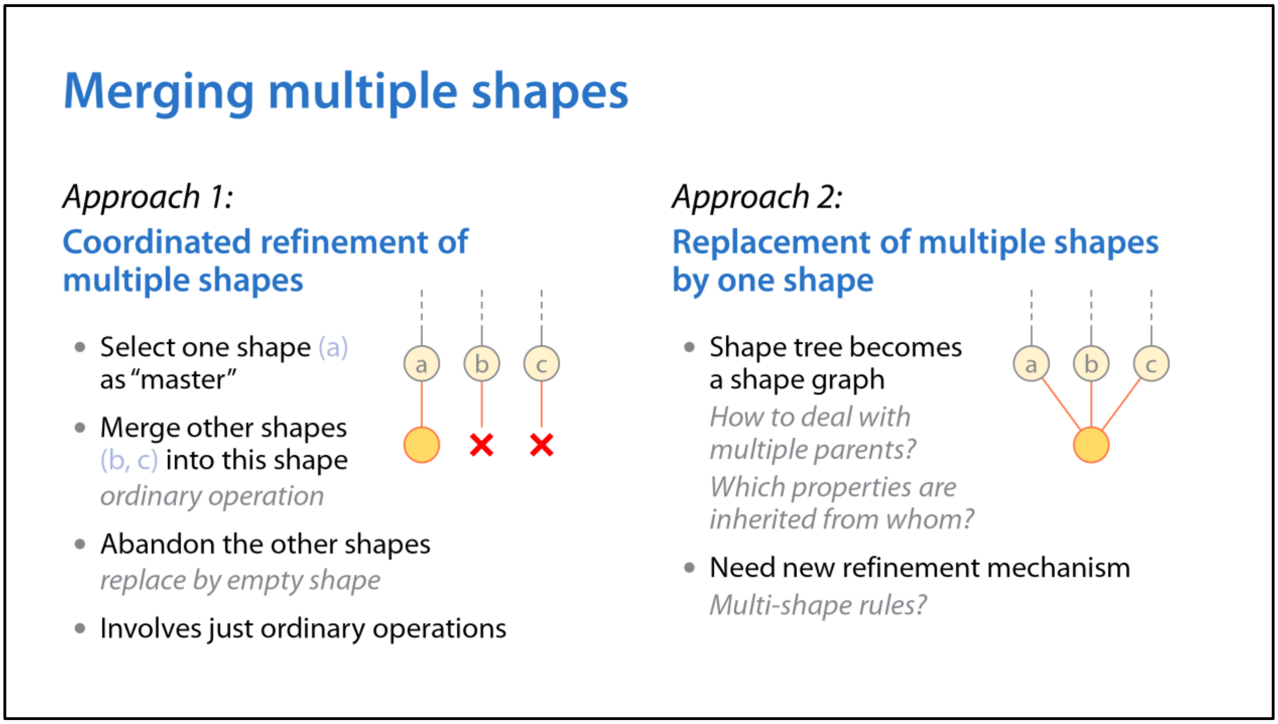
- Just a local refinement of the shape
- Ordinary operation, taking other shapes as argument: `minus(otherShapes)` (CGA++)

Establish connection between two shapes

- Select one shape as initiator, establish connection from it to other shape
- Becomes part of the refinement of one shape again
- Ordinary operation, taking other shape as argument: `connectTo(targetShape)` (CGA++)

Merge multiple shapes via Boolean union

- Affects refinement of all involved shapes



Merging multiple shapes

Approach 1:

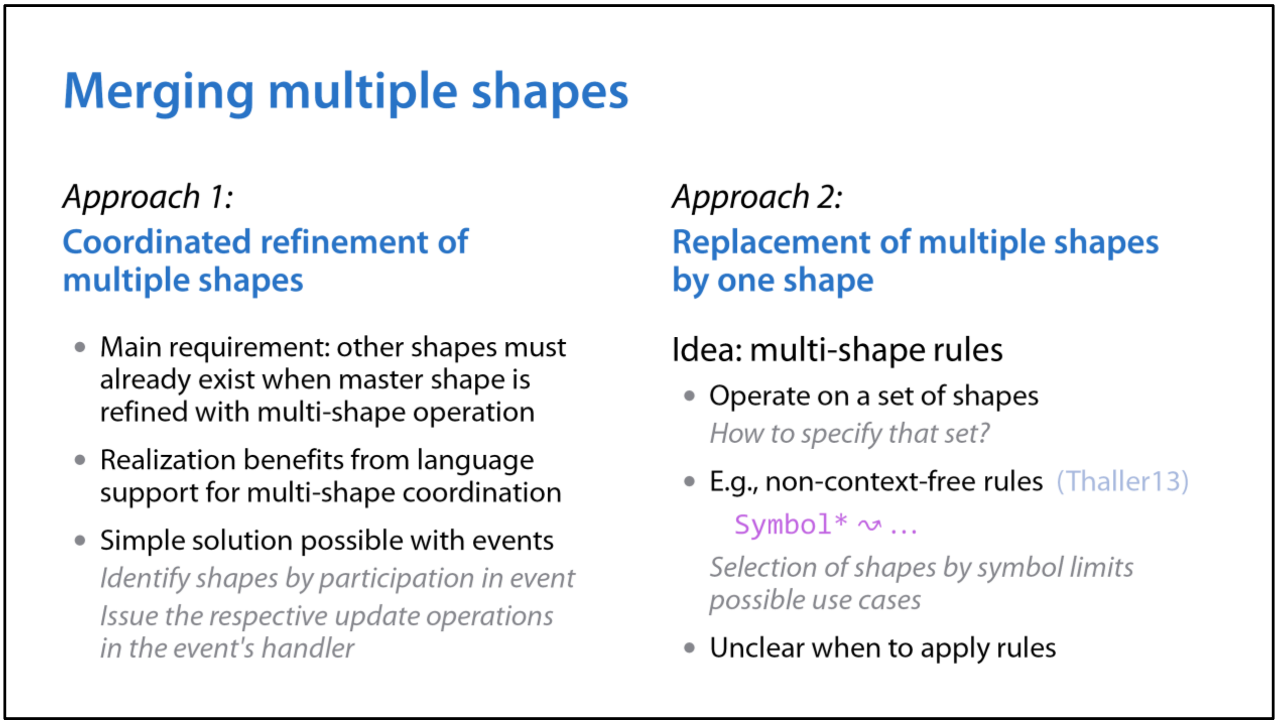
Coordinated refinement of multiple shapes

- Select one shape (a) as “master”
- Merge other shapes (b, c) into this shape
  ordinary operation
- Abandon the other shapes
  replace by empty shape
- Involves just ordinary operations

Approach 2:

Replacement of multiple shapes by one shape

- Shape tree becomes a shape graph
  How to deal with multiple parents?
  Which properties are inherited from whom?
- Need new refinement mechanism
  Multi-shape rules?



Merging multiple shapes

Approach 1:

Coordinated refinement of multiple shapes

- Main requirement: other shapes must already exist when master shape is refined with multi-shape operation
- Realization benefits from language support for multi-shape coordination
- Simple solution possible with events

  Identify shapes by participation in event

  Issue the respective update operations in the event's handler

Approach 2:

Replacement of multiple shapes by one shape

Idea: multi-shape rules

- Operate on a set of shapes

  How to specify that set?

- E.g., non-context-free rules (Thaller13)

  `Symbol* ↝ ...`

  Selection of shapes by symbol limits possible use cases

- Unclear when to apply rules

Multi-shape operations

Boolean operations (CGA++)

- `union(`*shape(s)*`)`
- `intersect(`*shape(s)*`)`
- `minus(`*shape(s)*`)`

create multiple connections at a time

current shape plays no role

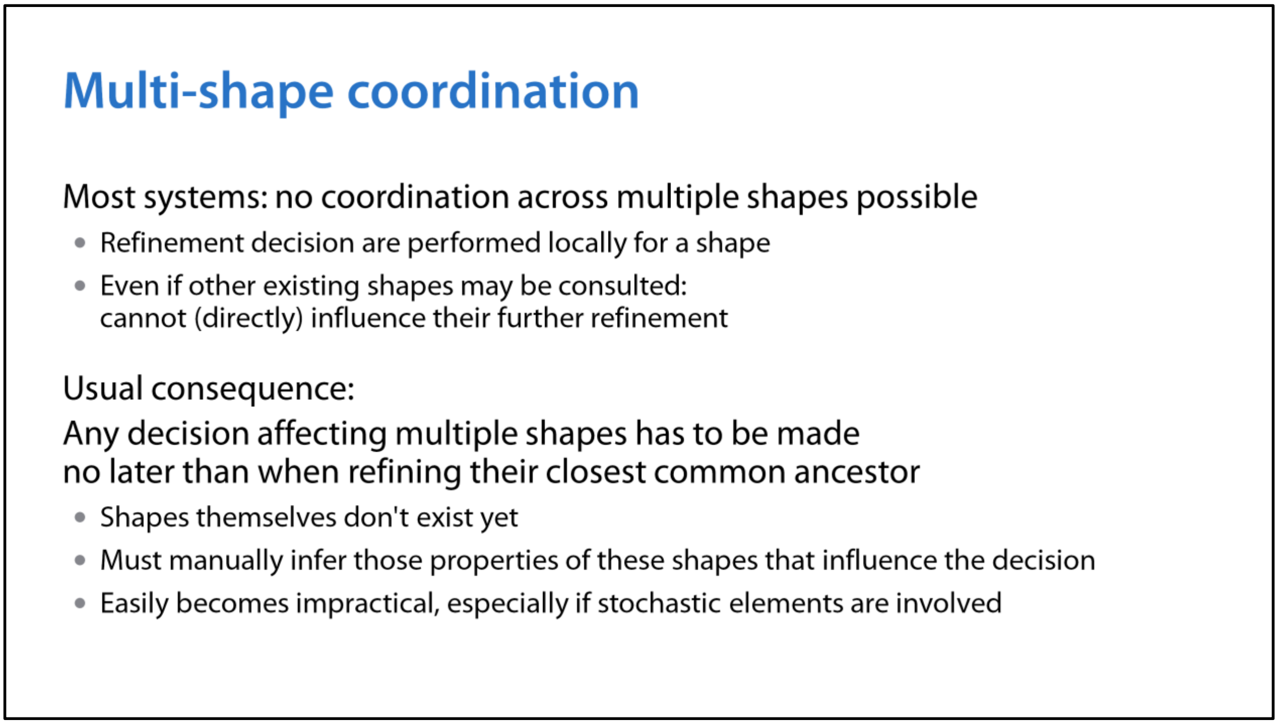
motivated by underlying derivation process

Adding interconnections

- `connectTo(`*targetShape*`)` (CGA++)
  create a connecting tube to the target shape; both shapes must be polygons

list of pairs of rectangles

- `beam(`*correspondences*`,` *rule*`,` *stiffness*`,` *gravity*`,` *step*`,` *threshold*`)` (Krecklau11)
  connect pairs of rectangles, creating a deformable beam for each
- `chain(`*correspondences*`,` *segments*`)` (Krecklau11)
  connect pairs of rectangles, creating a rigid chain for each



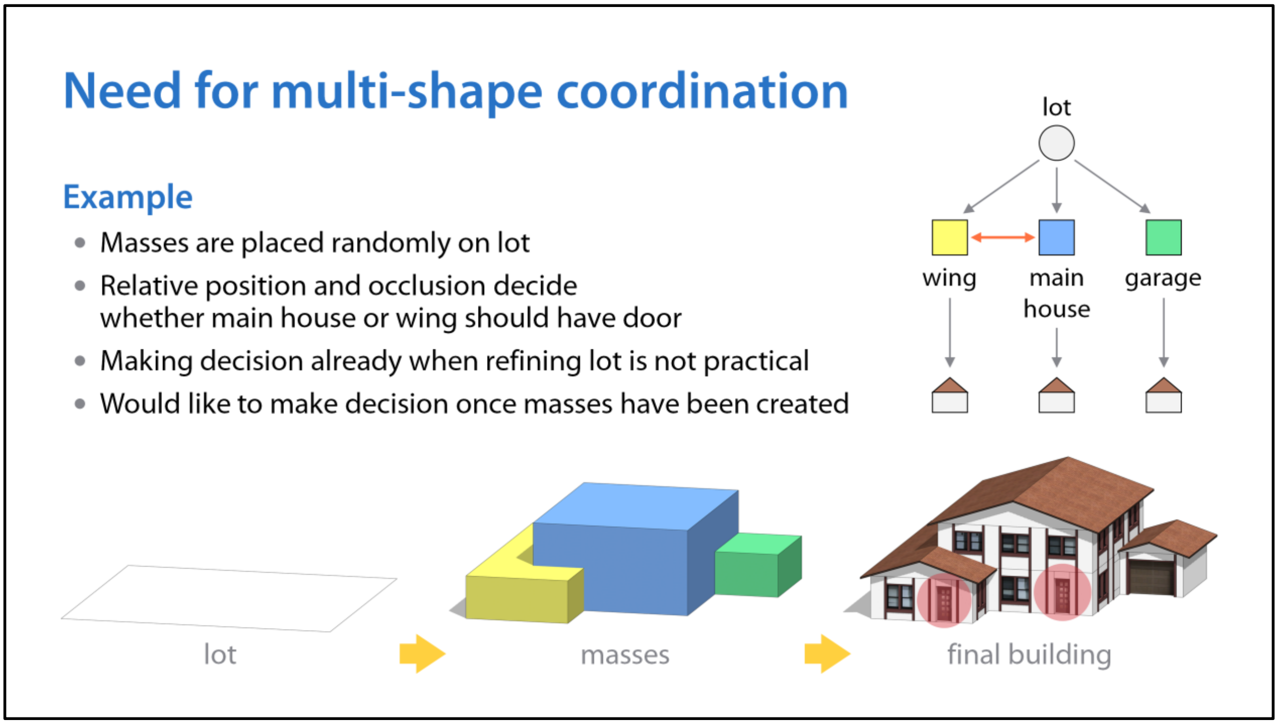
Multi-shape coordination

Most systems: no coordination across multiple shapes possible

- Refinement decision are performed locally for a shape
- Even if other existing shapes may be consulted:
  cannot (directly) influence their further refinement

Usual consequence:
Any decision affecting multiple shapes has to be made
no later than when refining their closest common ancestor

- Shapes themselves don't exist yet
- Must manually infer those properties of these shapes that influence the decision
- Easily becomes impractical, especially if stochastic elements are involved



Need for multi-shape coordination

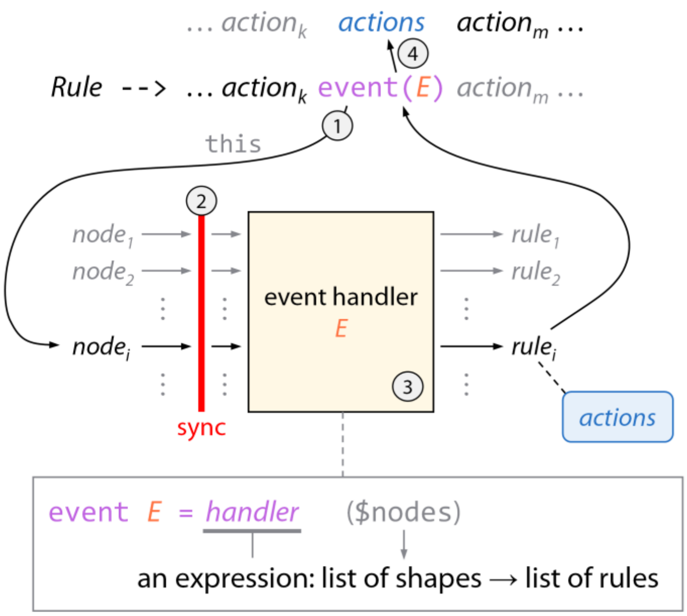
Example

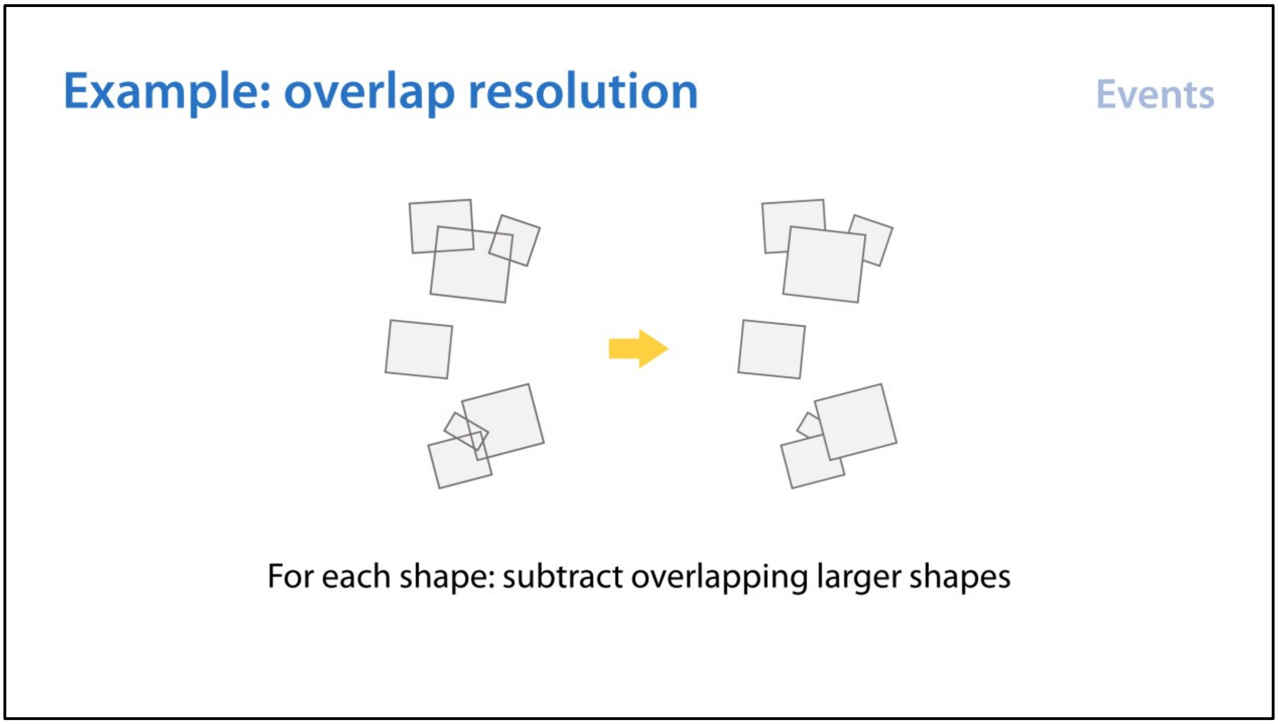
- Masses are placed randomly on lot
- Relative position and occlusion decide whether main house or wing should have door
- Making decision already when refining lot is not practical
- Would like to make decision once masses have been created

Events (CGA++)

Procedure

1. Operation `event` raises an event
   suspends current derivation branch
2. Wait until all derivation branches have raised some event
3. Consult event handler
   input: the current shapes of all participating branches
   outputs a rule for each branch
4. Resume derivation branch
   first executes the returned rule in-place

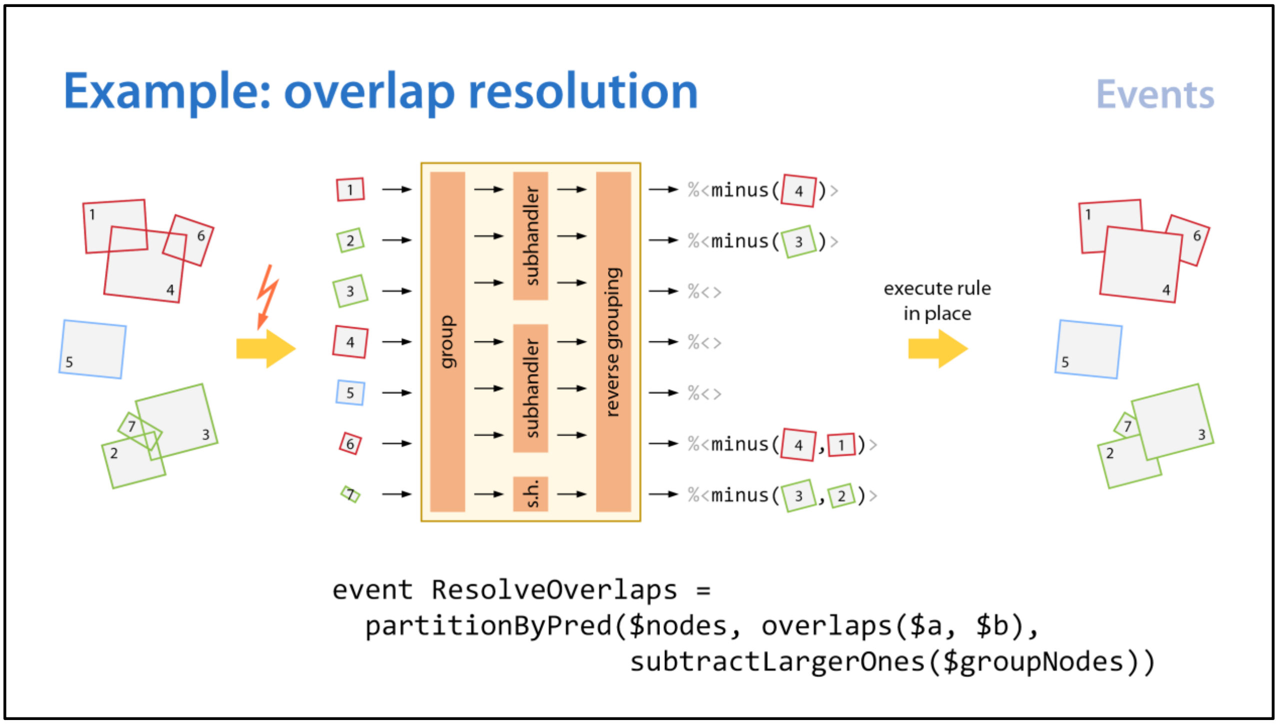




Example: overlap resolution

Events

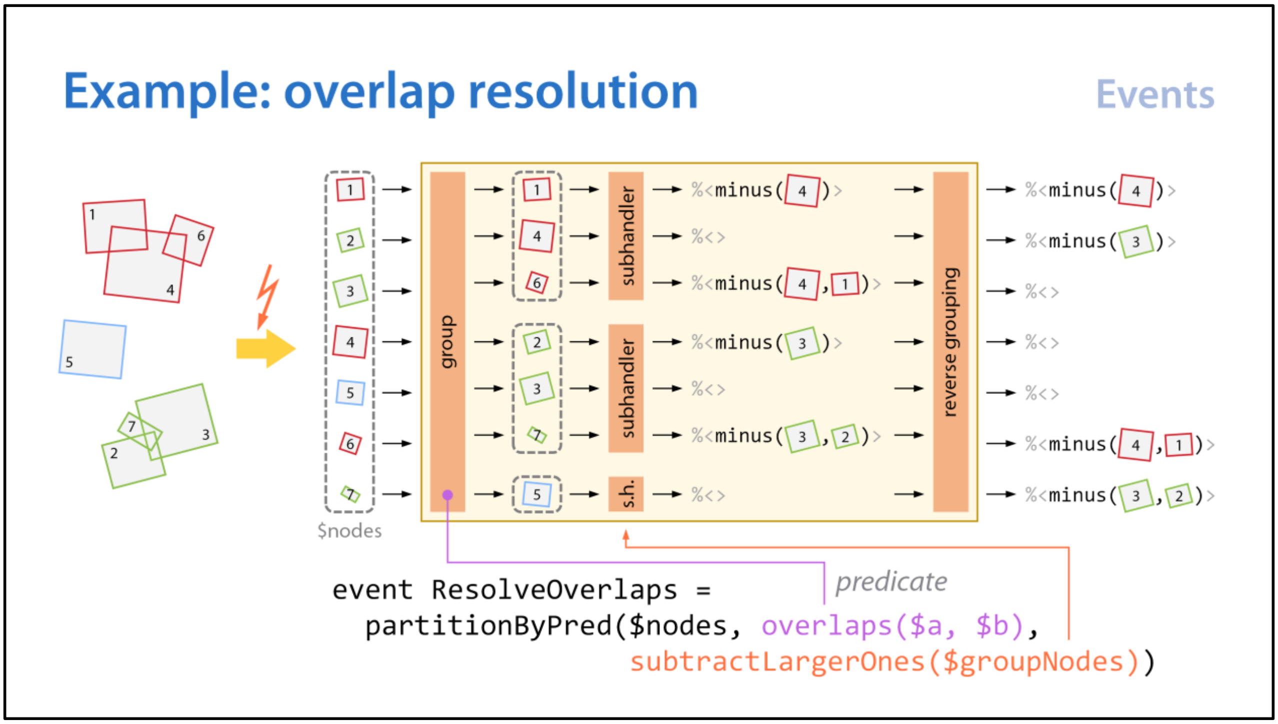
For each shape: subtract overlapping larger shapes



Example: overlap resolution

Events

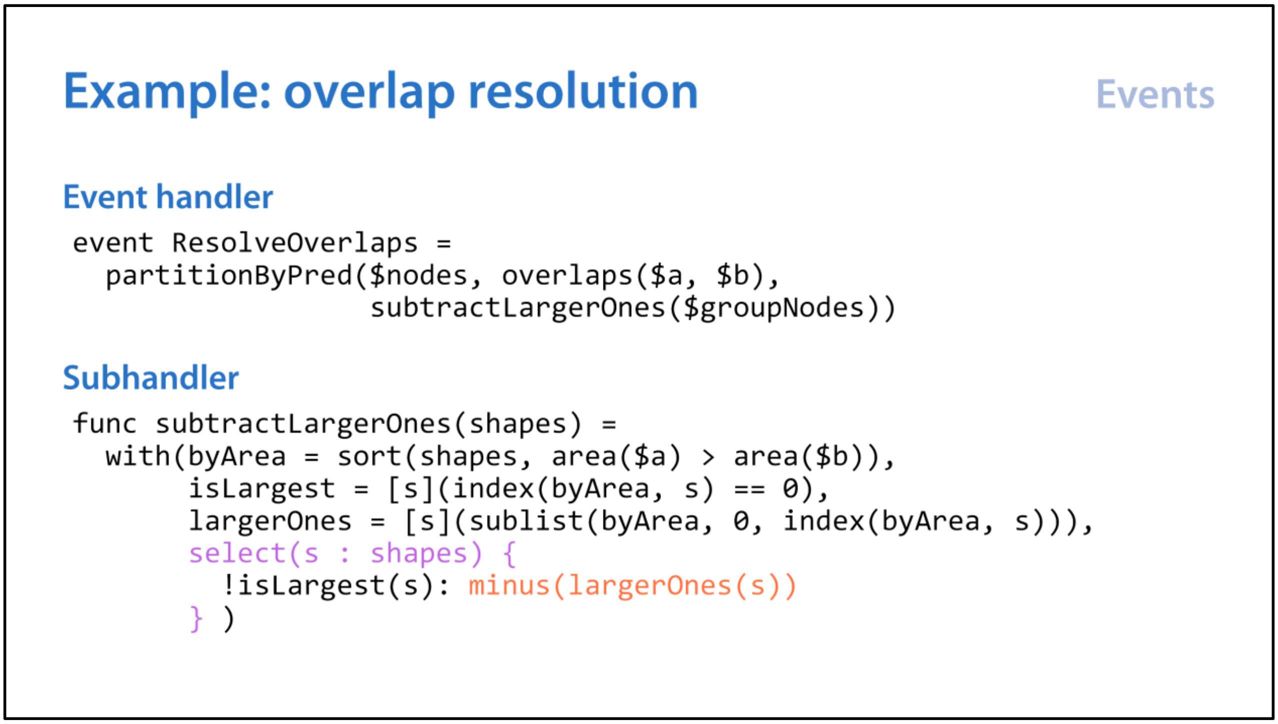
```
event ResolveOverlaps =
  partitionByPred($nodes, overlaps($a, $b),
                  subtractLargerOnes($groupNodes))
```



Example: overlap resolution

Events

```
event ResolveOverlaps =
  partitionByPred($nodes, overlaps($a, $b),
                           subtractLargerOnes($groupNodes))
```



Example: overlap resolution

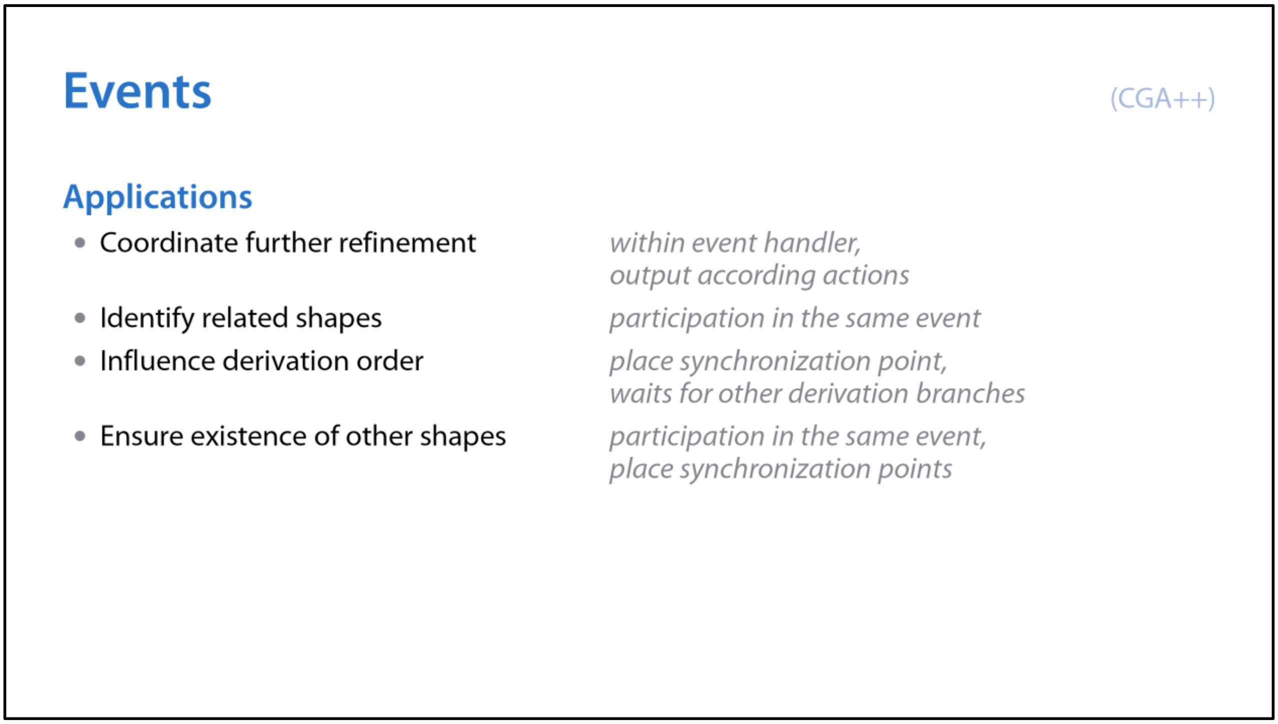
Events

Event handler

```
event ResolveOverlaps =
  partitionByPred($nodes, overlaps($a, $b),
                  subtractLargerOnes($groupNodes))
```

Subhandler

```
func subtractLargerOnes(shapes) =
  with(byArea = sort(shapes, area($a) > area($b)),
       isLargest = [s](index(byArea, s) == 0),
       largerOnes = [s](sublist(byArea, 0, index(byArea, s))),
       select(s : shapes) {
         !isLargest(s): minus(largerOnes(s))
       } )
```

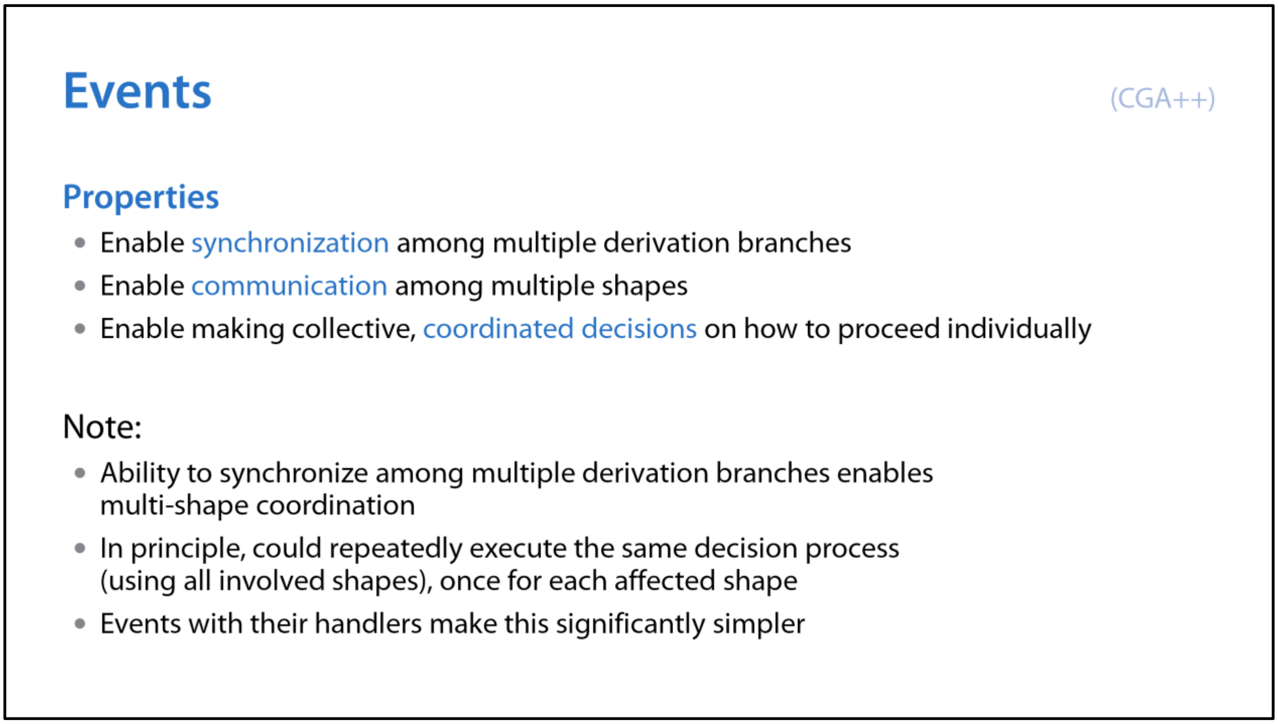


Events

(CGA++)

Applications

Coordinate further refinement	*within event handler,* *output according actions*
Identify related shapes	*participation in the same event*
Influence derivation order	*place synchronization point,* *waits for other derivation branches*
Ensure existence of other shapes	*participation in the same event,* *place synchronization points*



Events

(CGA++)

Properties

- Enable synchronization among multiple derivation branches
- Enable communication among multiple shapes
- Enable making collective, coordinated decisions on how to proceed individually

Note:

- Ability to synchronize among multiple derivation branches enables multi-shape coordination
- In principle, could repeatedly execute the same decision process (using all involved shapes), once for each affected shape
- Events with their handlers make this significantly simpler

Event handler

(CGA++)

Characteristics

- Arbitrary expression, yielding a list of rules (of same size as the input list of shapes)
- Facilitates reuse & compositing
- Enables dynamic grouping and hierarchical handling

Convenience handler functions

- Offer a concise syntax for common use cases

-
  ```
  select(s : shapes) {
     condition1 : actions1
   | condition2 = subhandler2
   | ...
  }
  ```

index of "master" shape

-
  ```
  forall(shapes, "union", 0) {
     postUnionActions
  }
  ```

-
  ```
  foreach(s : shapes) { actions }
  ```

Events: advanced features

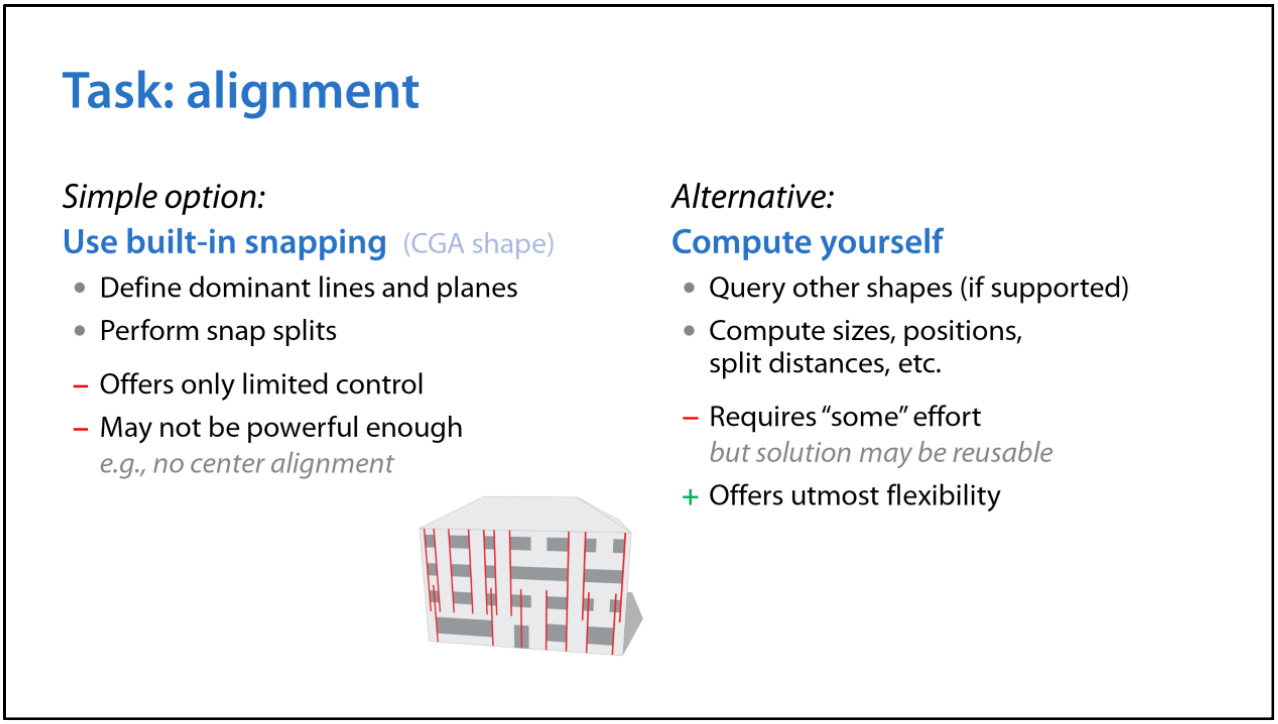
(CGA++)

Scoping mechanism: **event groups**

- Events operate globally by default,
  but can be made local to a subtree via event groups
- Operation `group(name)` creates a special group node;
  all shapes created by succeeding actions become descendants
- Specifying a group name when raising an event makes the event instance
  local to the subtree rooted in the closest matching group node ancestor
- Different instances of an event (e.g., one for each floor) can coexist

Signaling

- Handling an event indicates that a certain stage has been reached
- Operation `wait` allows waiting until an event got handled
  for the first time (within a certain subtree)



Task: alignment

Simple option:

Use built-in snapping (CGA shape)

- Define dominant lines and planes
- Perform snap splits

– Offers only limited control

– May not be powerful enough
e.g., no center alignment

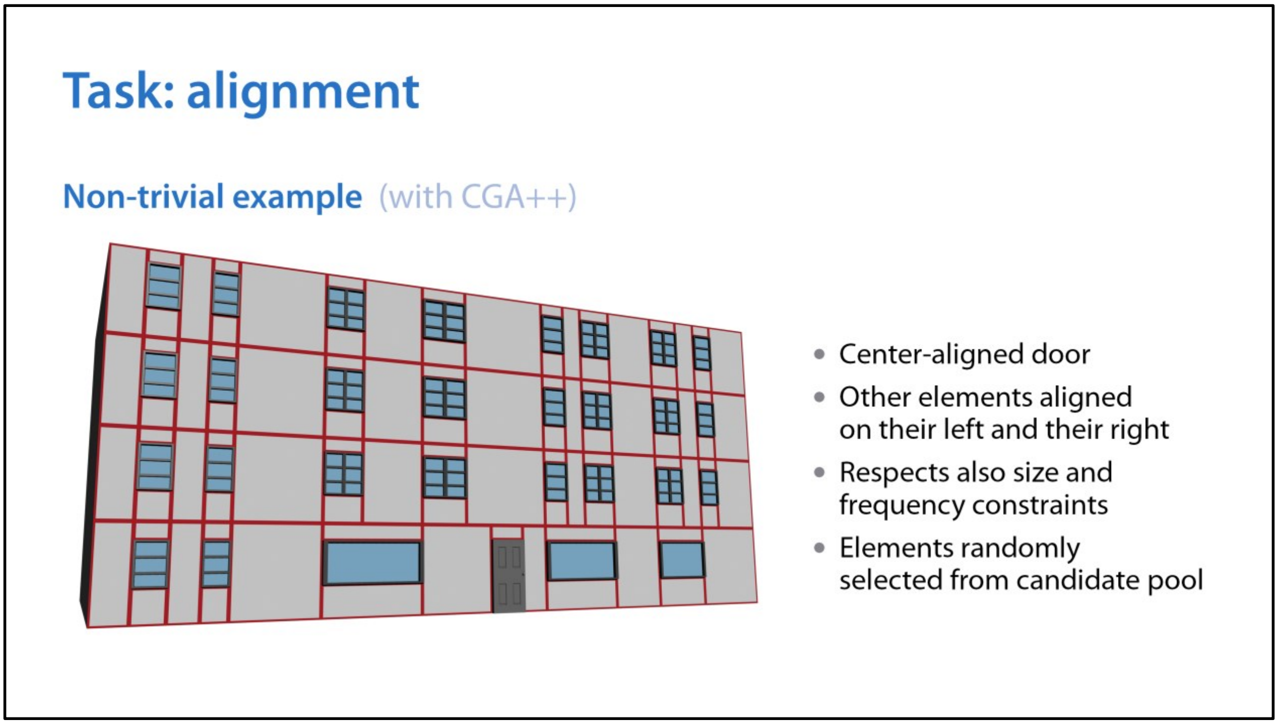
Alternative:

Compute yourself

- Query other shapes (if supported)
- Compute sizes, positions,
  split distances, etc.

– Requires "some" effort
but solution may be reusable

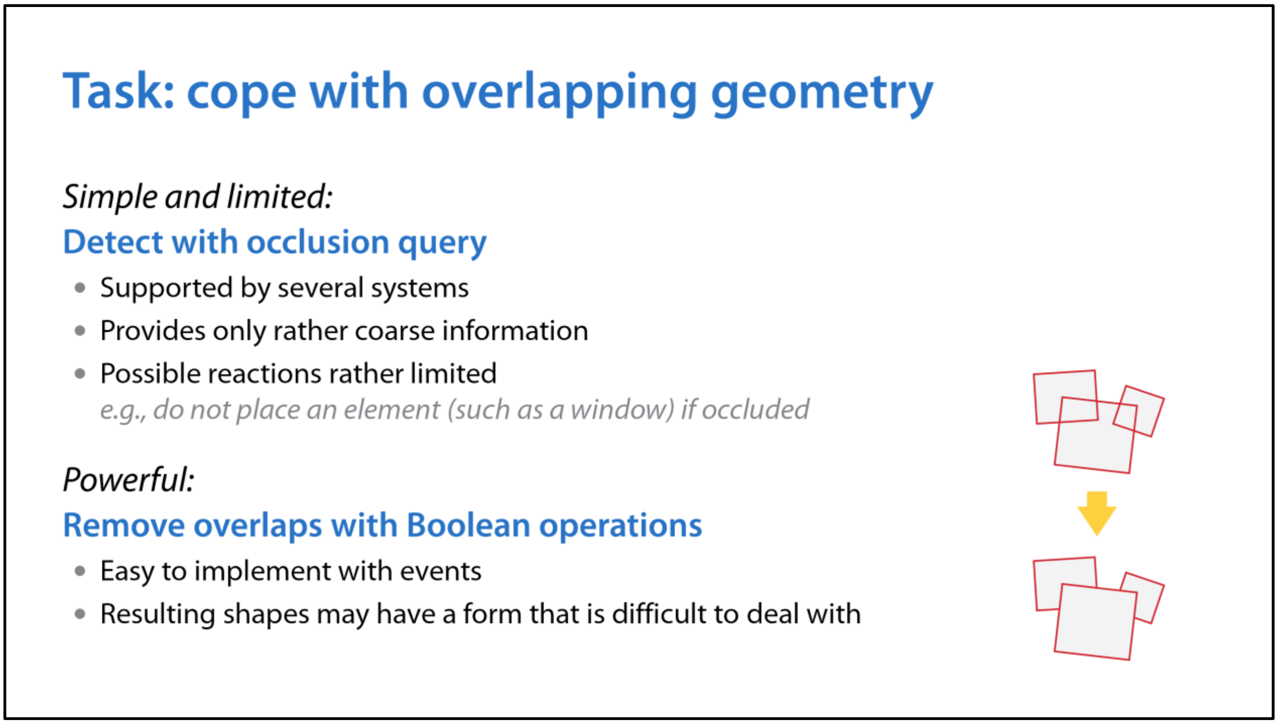
+ Offers utmost flexibility



Task: alignment

Non-trivial example (with CGA++)

- Center-aligned door
- Other elements aligned on their left and their right
- Respects also size and frequency constraints
- Elements randomly selected from candidate pool



Task: cope with overlapping geometry

Simple and limited:

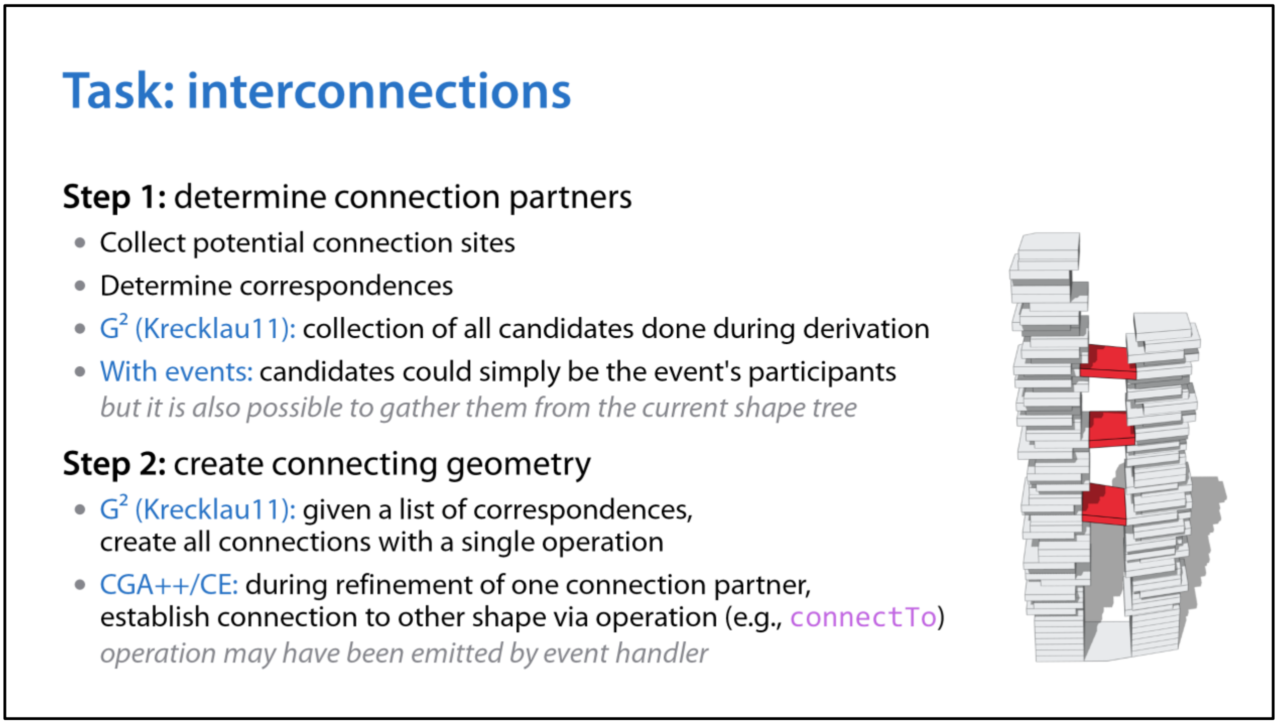
Detect with occlusion query

- Supported by several systems
- Provides only rather coarse information
- Possible reactions rather limited
  e.g., do not place an element (such as a window) if occluded

Powerful:

Remove overlaps with Boolean operations

- Easy to implement with events
- Resulting shapes may have a form that is difficult to deal with



Task: interconnections

Step 1: determine connection partners

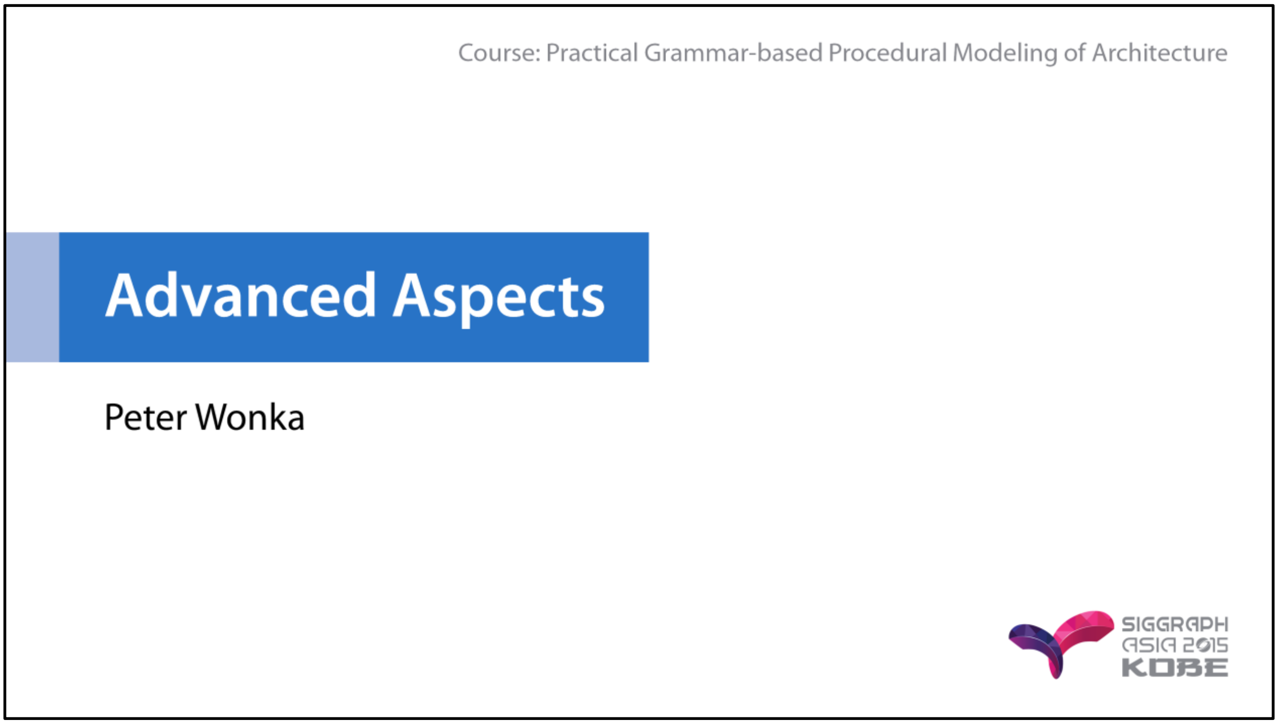
- Collect potential connection sites
- Determine correspondences
- G^2 (Krecklau11): collection of all candidates done during derivation
- With events: candidates could simply be the event's participants
  but it is also possible to gather them from the current shape tree

Step 2: create connecting geometry

- G^2 (Krecklau11): given a list of correspondences,
  create all connections with a single operation
- CGA++/CE: during refinement of one connection partner,
  establish connection to other shape via operation (e.g., `connectTo`)
  operation may have been emitted by event handler

Examples: Favela project (external solution), Krecklau11, Schwarz15

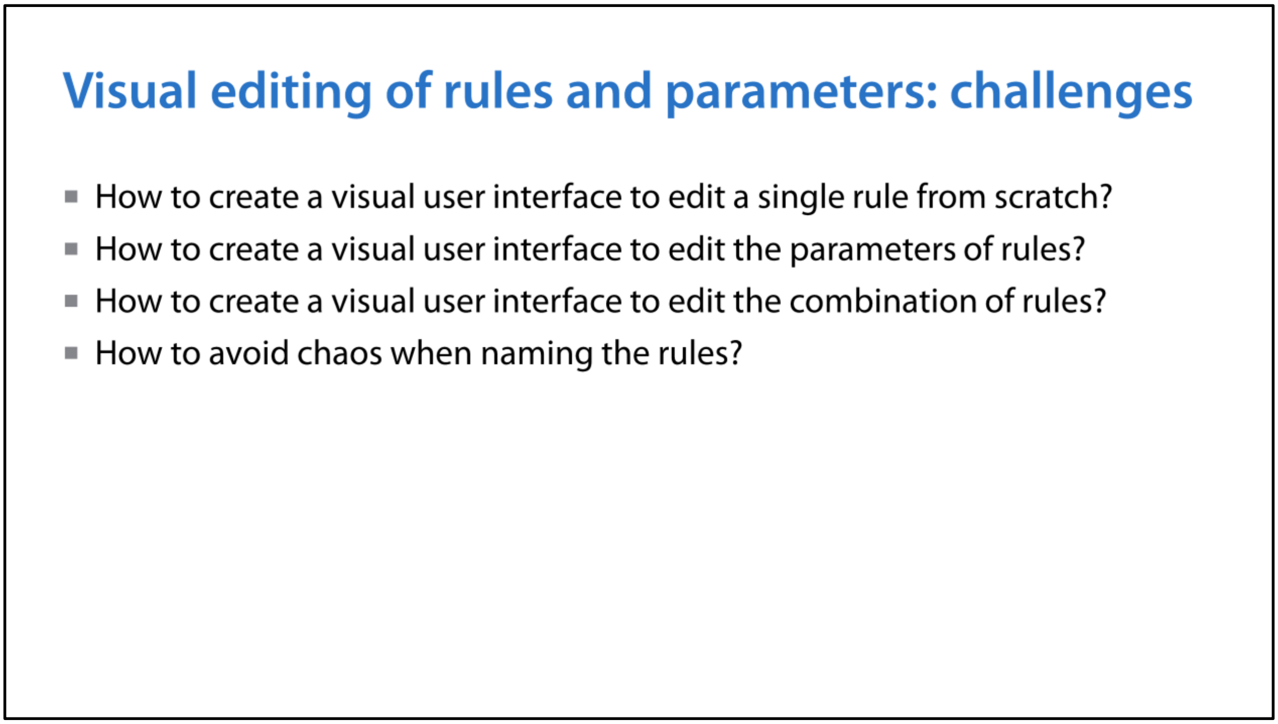
5 Advanced aspects



Course: Practical Grammar-based Procedural Modeling of Architecture

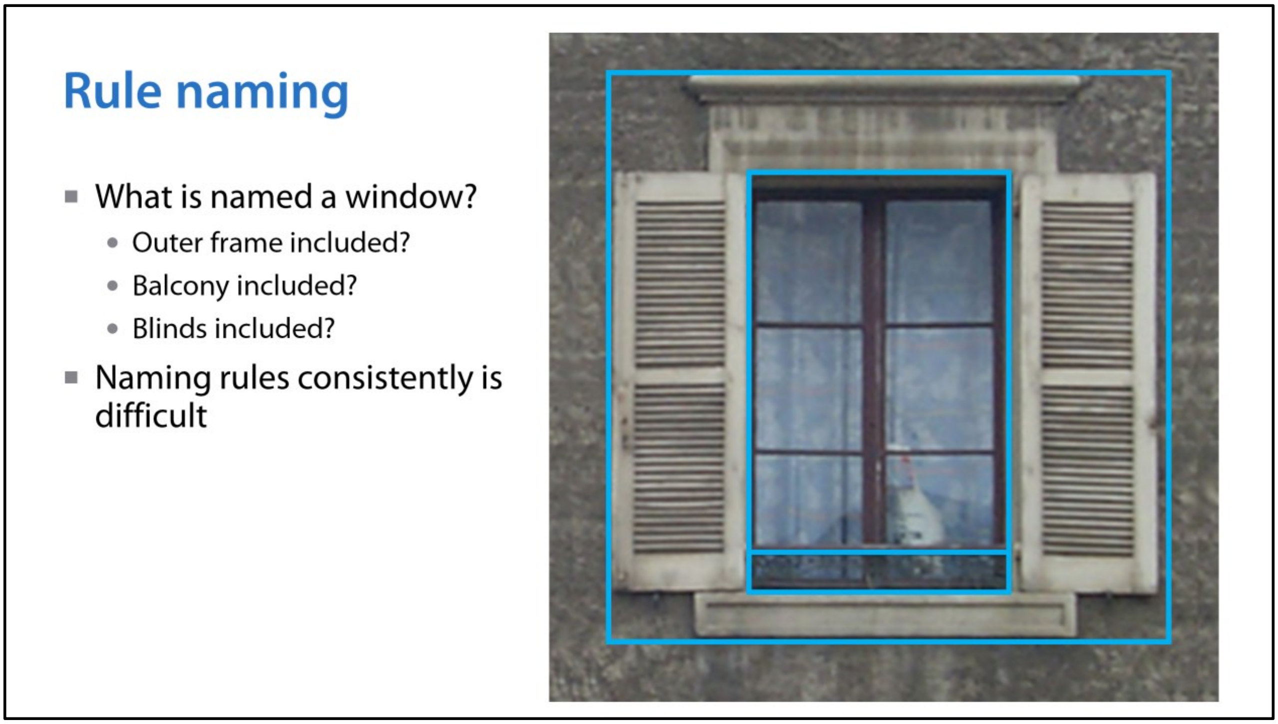
Advanced Aspects

Peter Wonka



Visual editing of rules and parameters: challenges

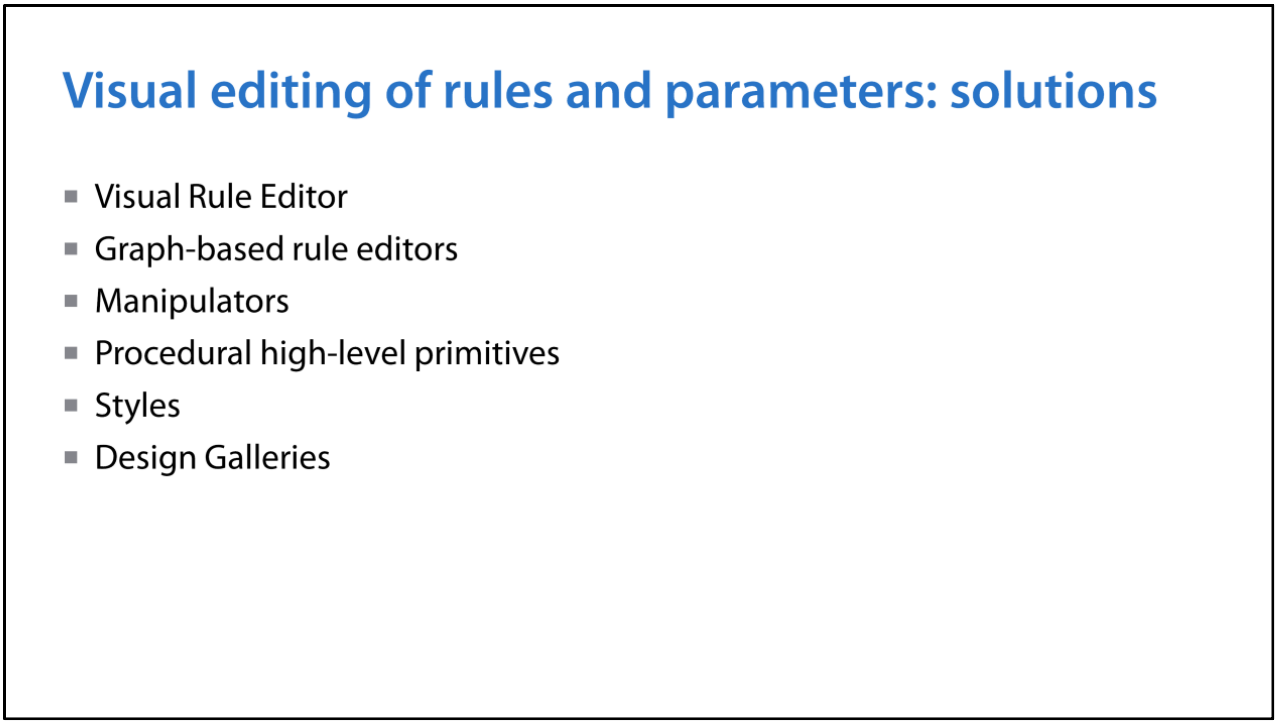
- How to create a visual user interface to edit a single rule from scratch?
- How to create a visual user interface to edit the parameters of rules?
- How to create a visual user interface to edit the combination of rules?
- How to avoid chaos when naming the rules?



Rule naming

- What is named a window?
  - Outer frame included?
  - Balcony included?
  - Blinds included?
- Naming rules consistently is difficult

Image: Wonka



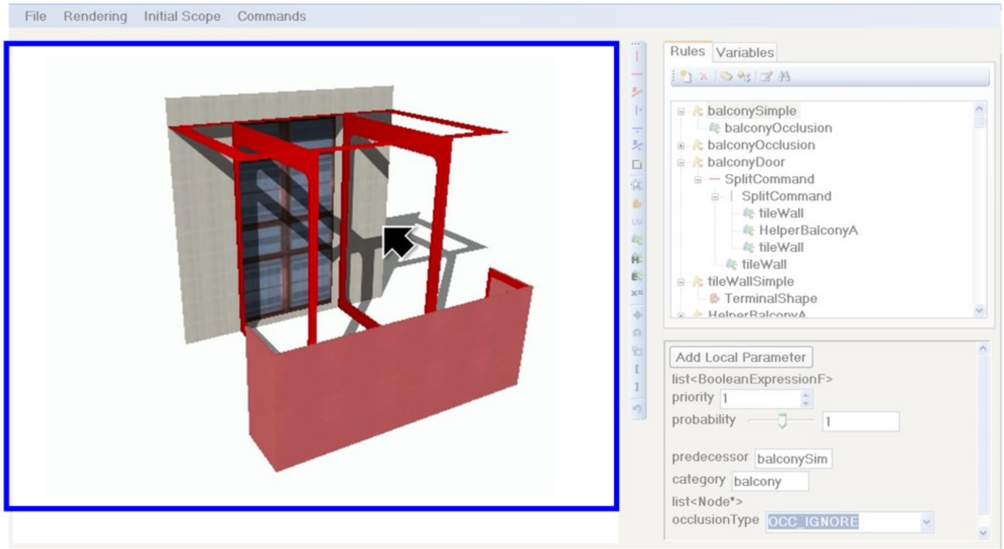
Visual editing of rules and parameters: solutions

- Visual Rule Editor
- Graph-based rule editors
- Manipulators
- Procedural high-level primitives
- Styles
- Design Galleries

Visual rule editing

- Visual interface to define rules and their parameters
- Videos from Lipp2008

Interactive rule visualization



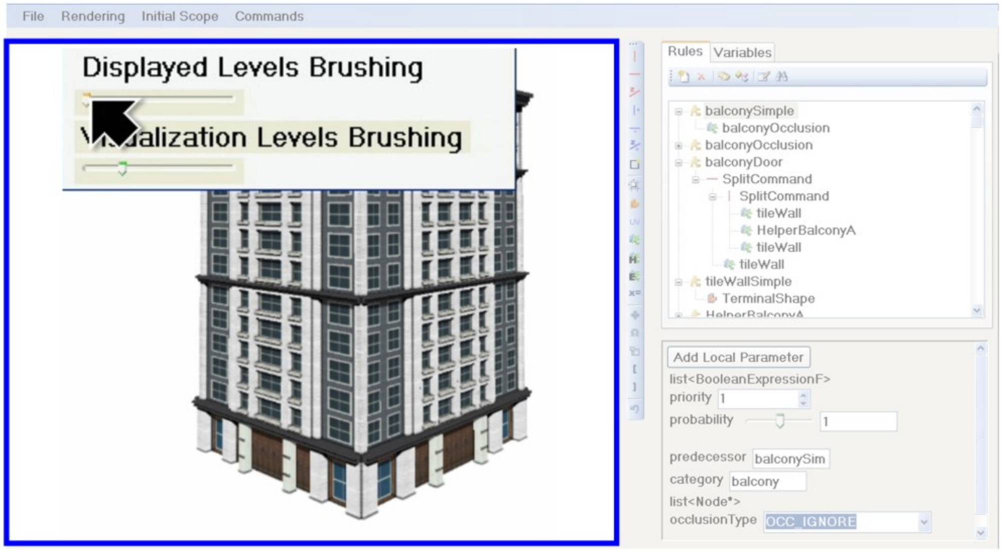
- balconySimple
  - balconyOcclusion
- balconyOcclusion
- balconyDoor
  - SplitCommand
    - SplitCommand
      - tileWall
      - HelperBalconyA
      - tileWall
    - tileWall
- tileWallSimple
  - TerminalShape
- HelperBalconyA

list<BooleanExpressionF>
priority 1
probability 1

predecessor balconySim
category balcony
list<Node*>
occlusionType OCC_IGNORE

Video: Lipp 2008

Focus & context editing



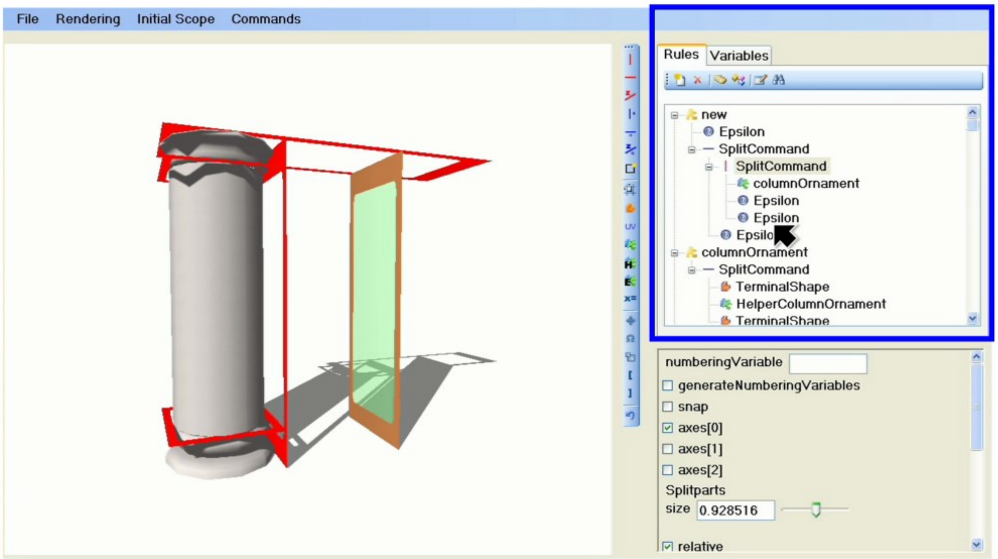
- balconySimple
  - balconyOcclusion
- balconyOcclusion
- balconyDoor
  - SplitCommand
    - SplitCommand
      - tileWall
      - HelperBalconyA
      - tileWall
    - tileWall
- tileWallSimple
  - TerminalShape
- HelperBalconyA

list<BooleanExpressionF>
priority 1
probability 1

predecessor balconySim
category balcony
list<Node*>
occlusionType OCC_IGNORE

Video: Lipp 2008

Linked tree-view: rule explorer



```
new
  Epsilon
  SplitCommand
    SplitCommand
      columnOrnament
      Epsilon
      Epsilon
    Epsilon
columnOrnament
  SplitCommand
    TerminalShape
    HelperColumnOrnament
    TerminalShape
```

numberingVariable

☐ generateNumberingVariables

☐ snap

☑ axes[0]

☐ axes[1]

☐ axes[2]

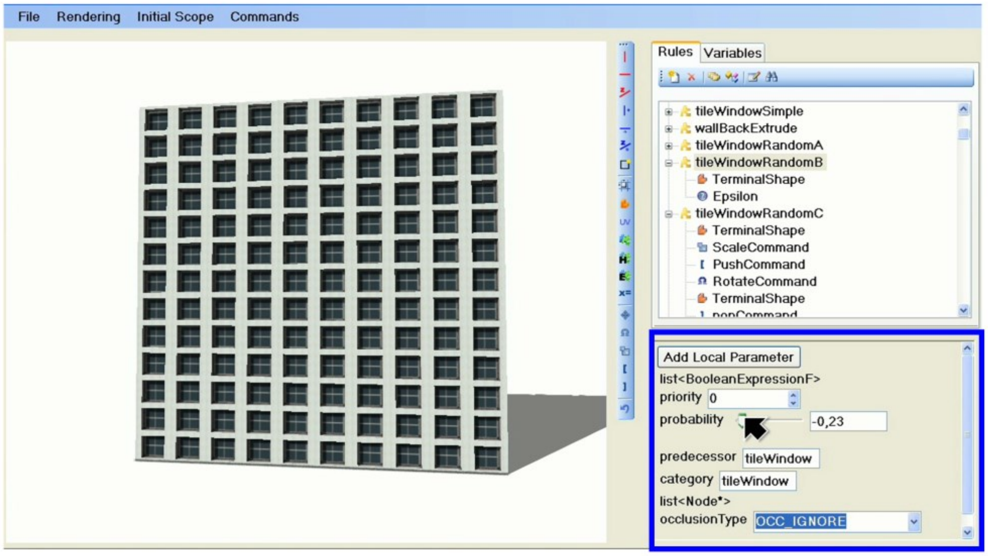
Splitparts

size 0.928516

☑ relative

Video: Lipp 2008

Parameter view

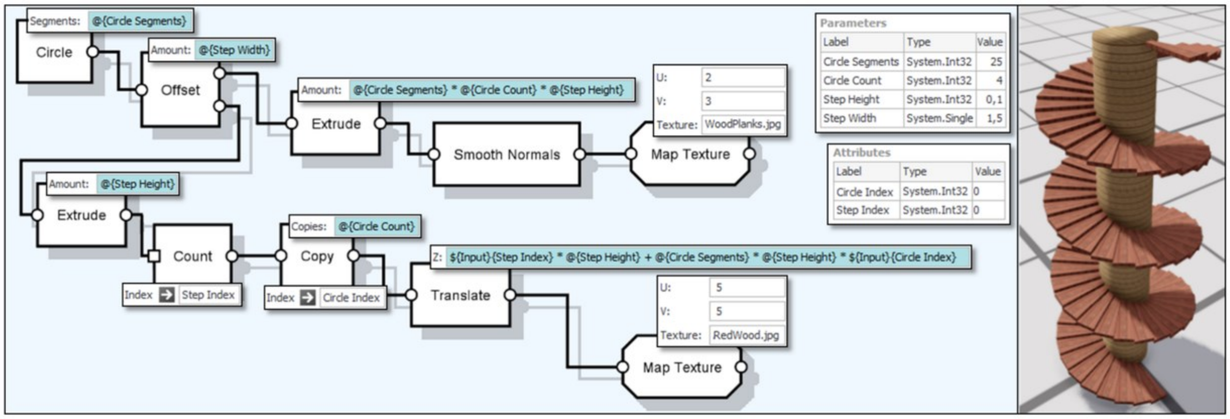


list<BooleanExpressionF>
priority 0
probability -0,23
predecessor tileWindow
category tileWindow
list<Node*>
occlusionType OCC_IGNORE

Video: Lipp 2008

Graph-based rule editors

- Examples: Silva2015, Thaller2012, Thaller2013, Patow2012, Houdini



Segments: @{Circle Segments}
Amount: @{Step Width}
Amount: @{Circle Segments} * @{Circle Count} * @{Step Height}
U: 2
V: 3
Texture: WoodPlanks.jpg
Amount: @{Step Height}
Copies: @{Circle Count}
Z: ${Input}{Step Index} * @{Step Height} + @{Circle Segments} * @{Step Height} * ${Input}{Circle Index}
Index → Step Index
Index → Circle Index
U: 5
V: 5
Texture: RedWood.jpg

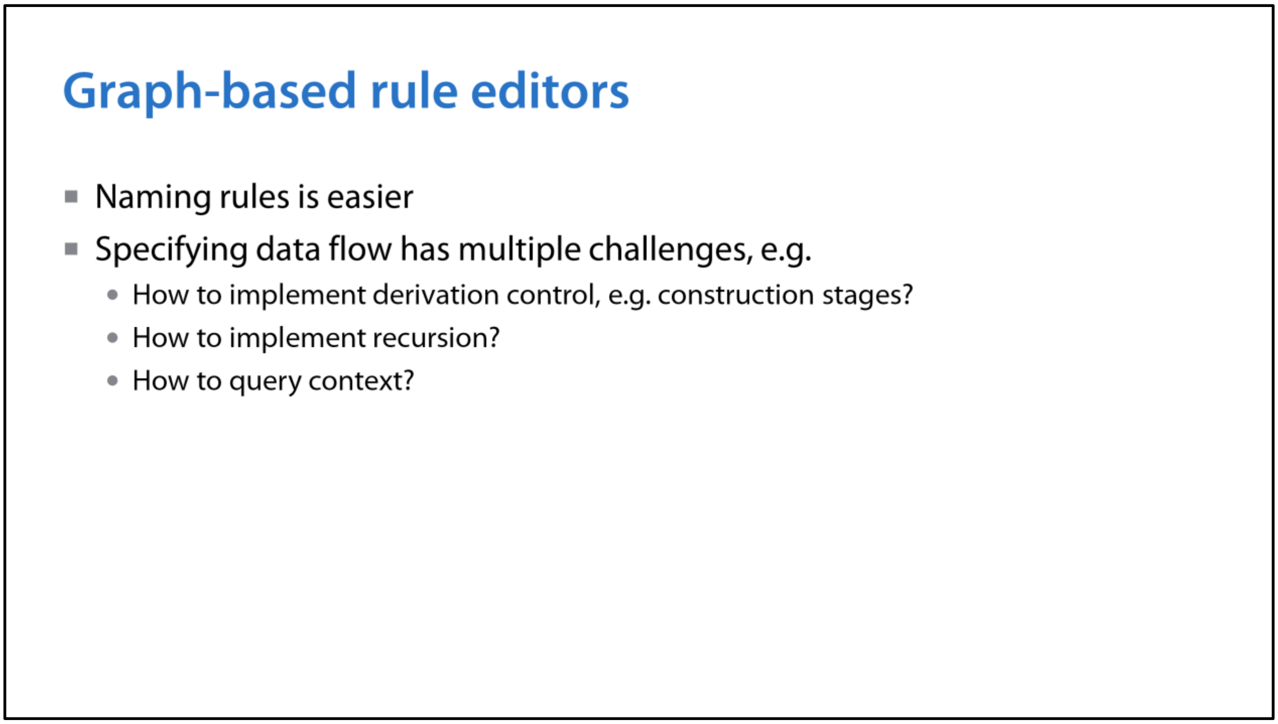
Parameters

Label	Type	Value
Circle Segments	System.Int32	25
Circle Count	System.Int32	4
Step Height	System.Int32	0,1
Step Width	System.Single	1,5

Attributes

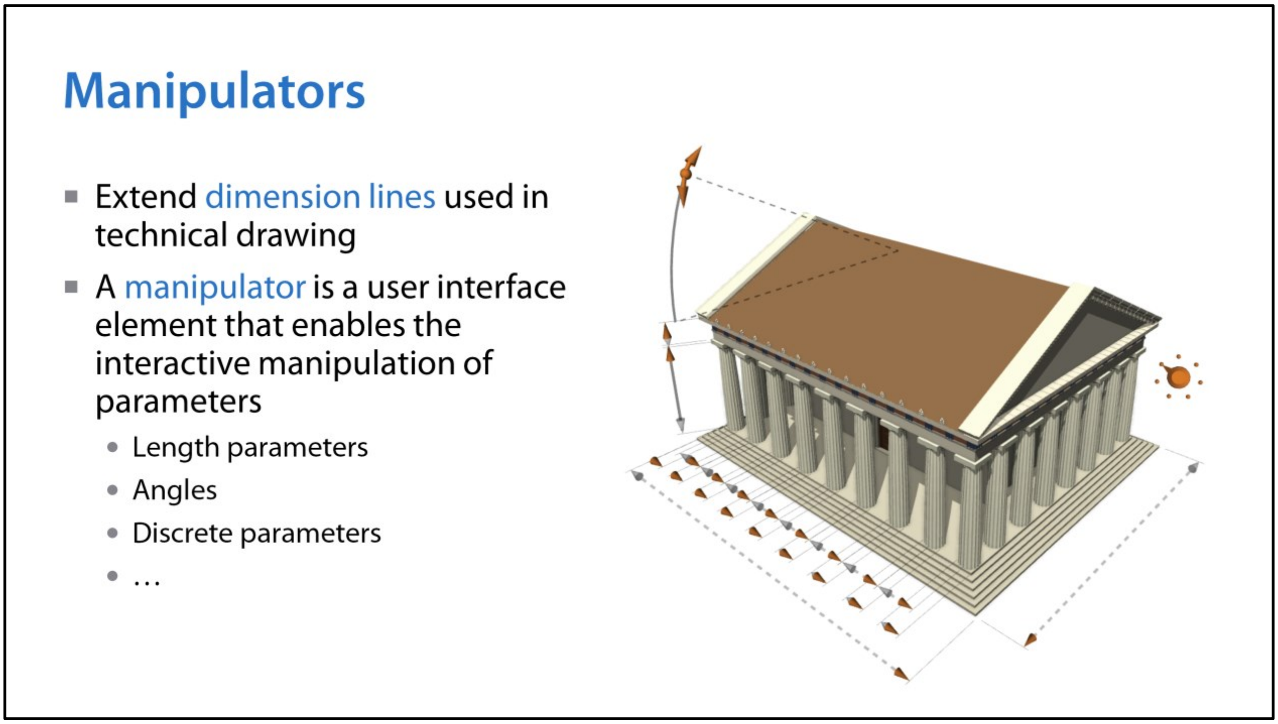
Label	Type	Value
Circle Index	System.Int32	0
Step Index	System.Int32	0

Image: Silva 2015



Graph-based rule editors

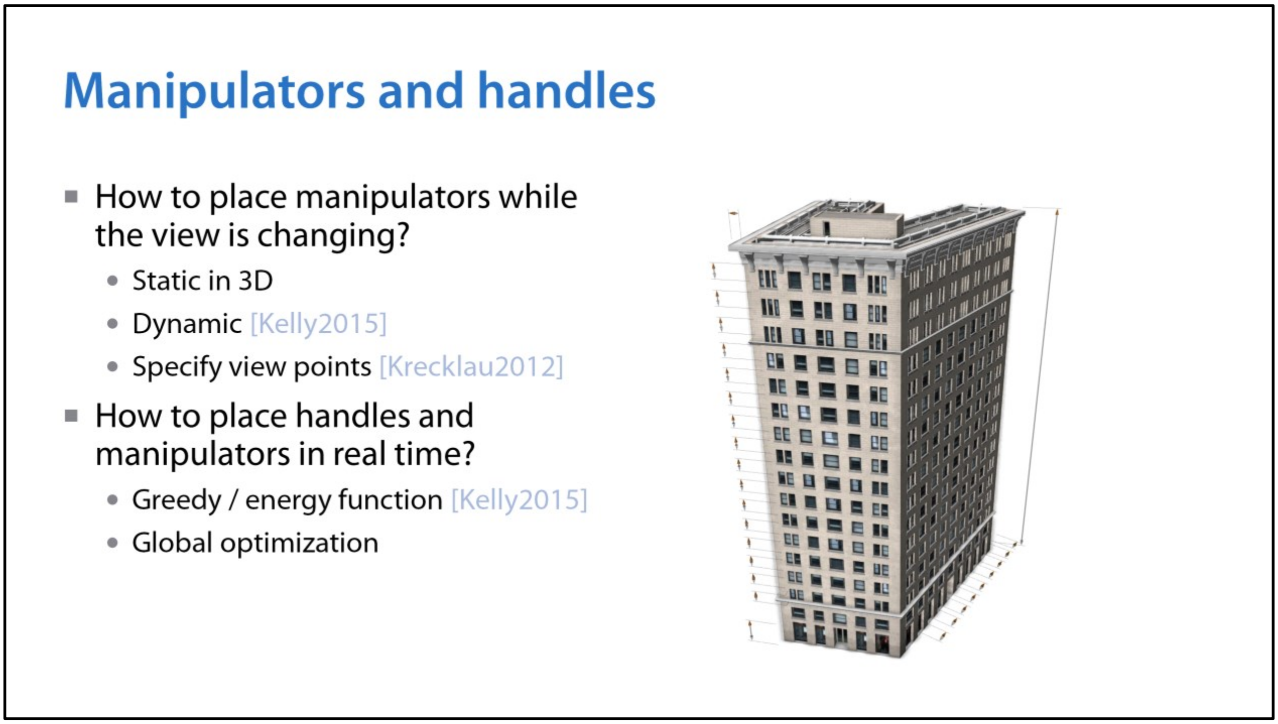
- Naming rules is easier
- Specifying data flow has multiple challenges, e.g.
    - How to implement derivation control, e.g. construction stages?
    - How to implement recursion?
    - How to query context?



Manipulators

- Extend dimension lines used in technical drawing
- A manipulator is a user interface element that enables the interactive manipulation of parameters
  - Length parameters
  - Angles
  - Discrete parameters
  - …

Image: Kelly 2015



Manipulators and handles

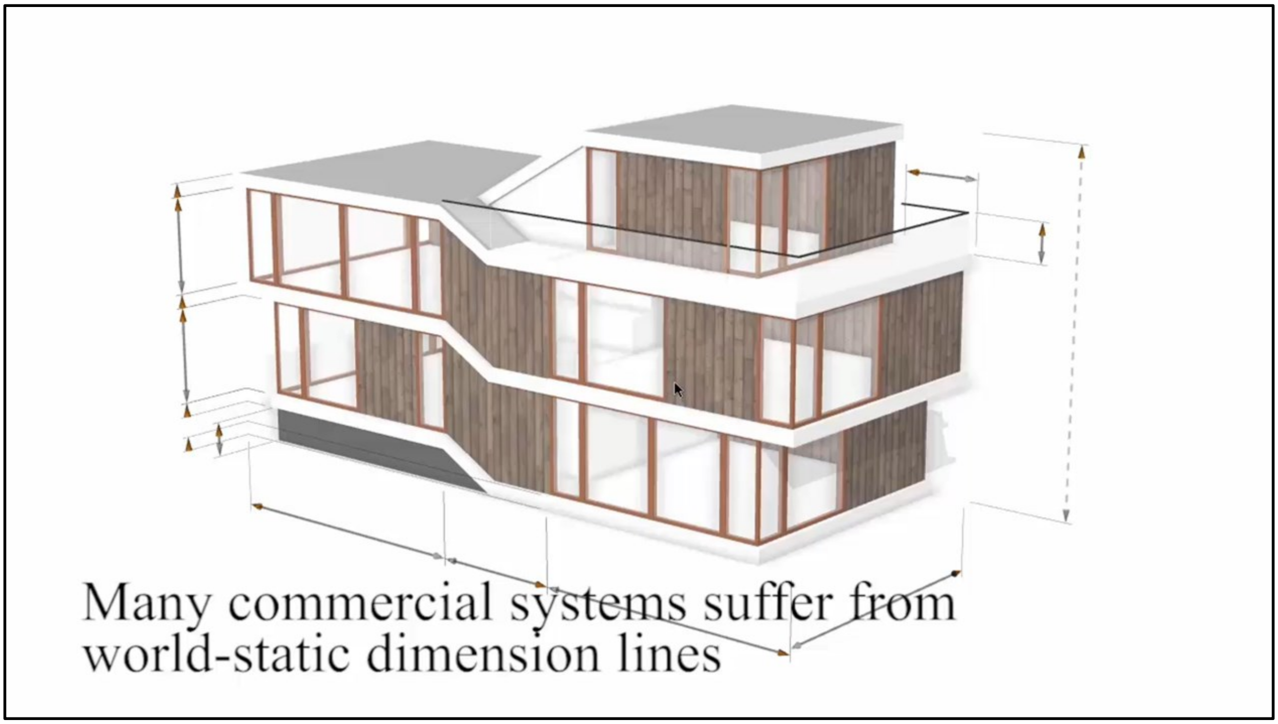
- How to place manipulators while the view is changing?
  - Static in 3D
  - Dynamic [Kelly2015]
  - Specify view points [Krecklau2012]
- How to place handles and manipulators in real time?
  - Greedy / energy function [Kelly2015]
  - Global optimization

Image: Kelly 2015

Manipulators and handles

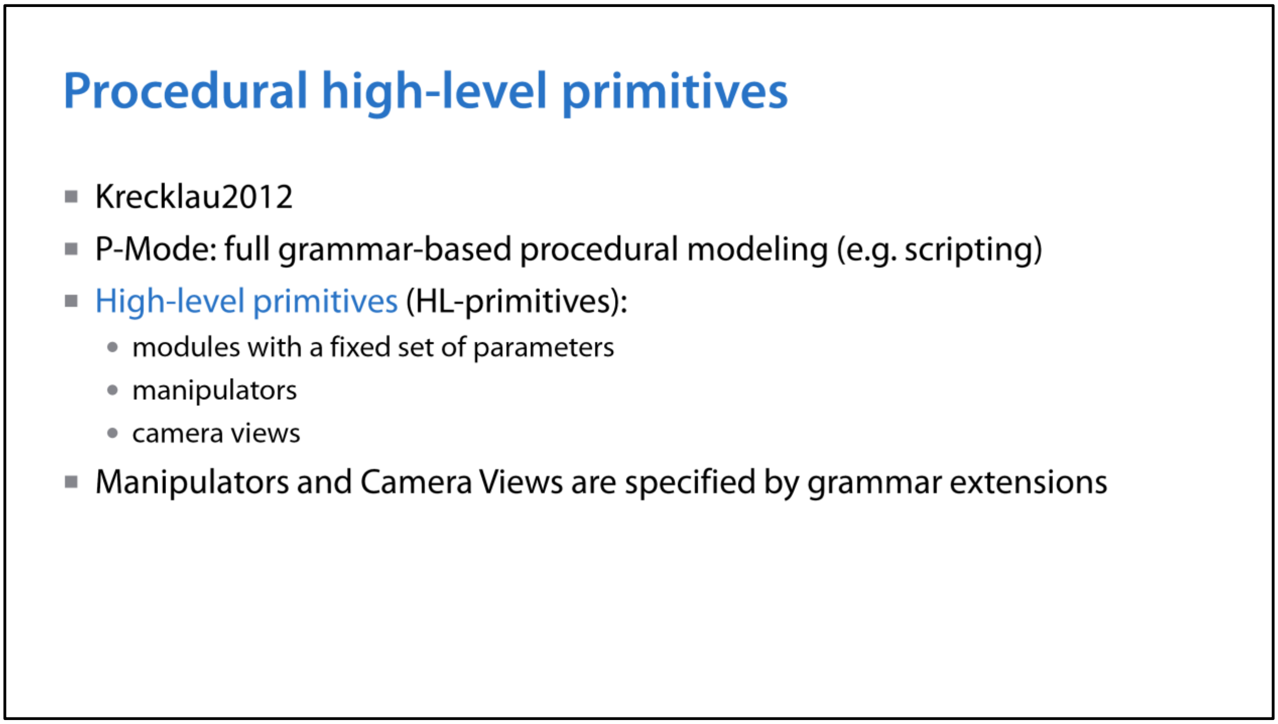
- How to specify placement parameters?
  - Automatically [Open Problem]
  - Manually in the grammar [Kelly2015, Krecklau2012]
- How to decide what parameters should have manipulators?
  - Automatically [Open Problem]
  - Manually in the grammar [Kelly2015, Krecklau2012]

Image: Kelly 2015



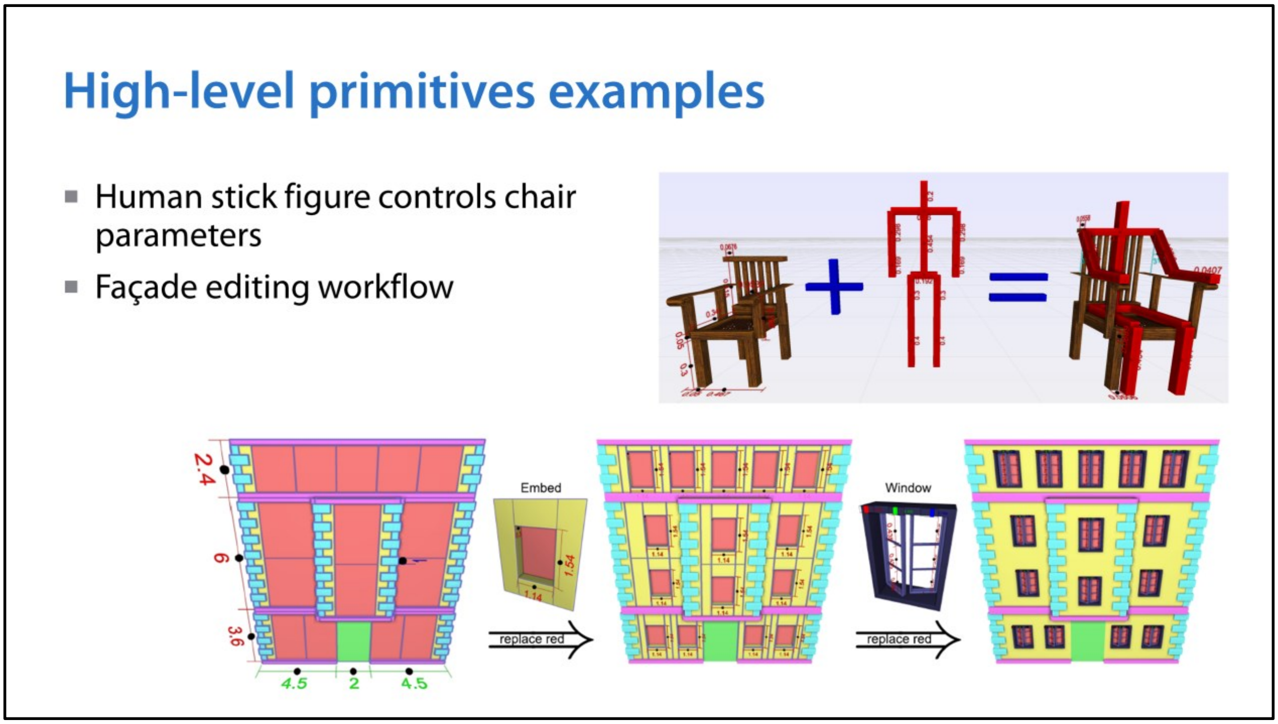
Many commercial systems suffer from world-static dimension lines

Video: Kelly 2015



Procedural high-level primitives

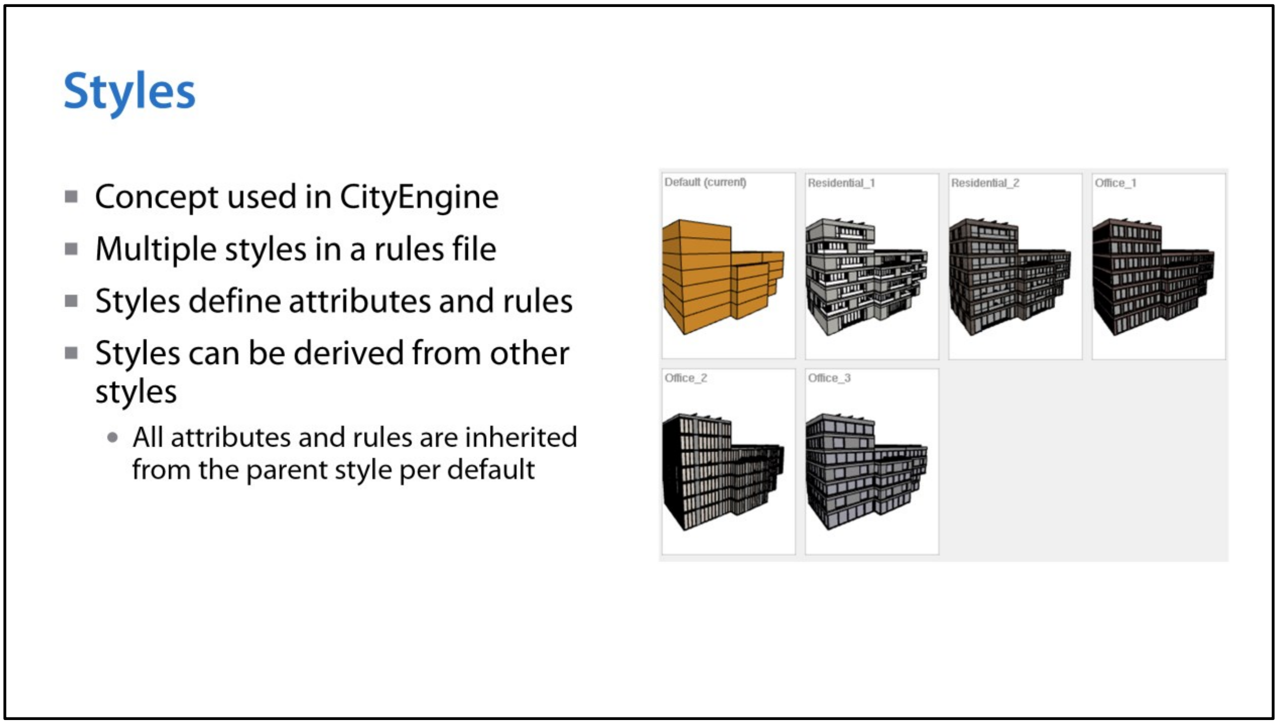
- Krecklau2012
- P-Mode: full grammar-based procedural modeling (e.g. scripting)
- High-level primitives (HL-primitives):
  - modules with a fixed set of parameters
  - manipulators
  - camera views
- Manipulators and Camera Views are specified by grammar extensions



High-level primitives examples

- Human stick figure controls chair parameters
- Façade editing workflow

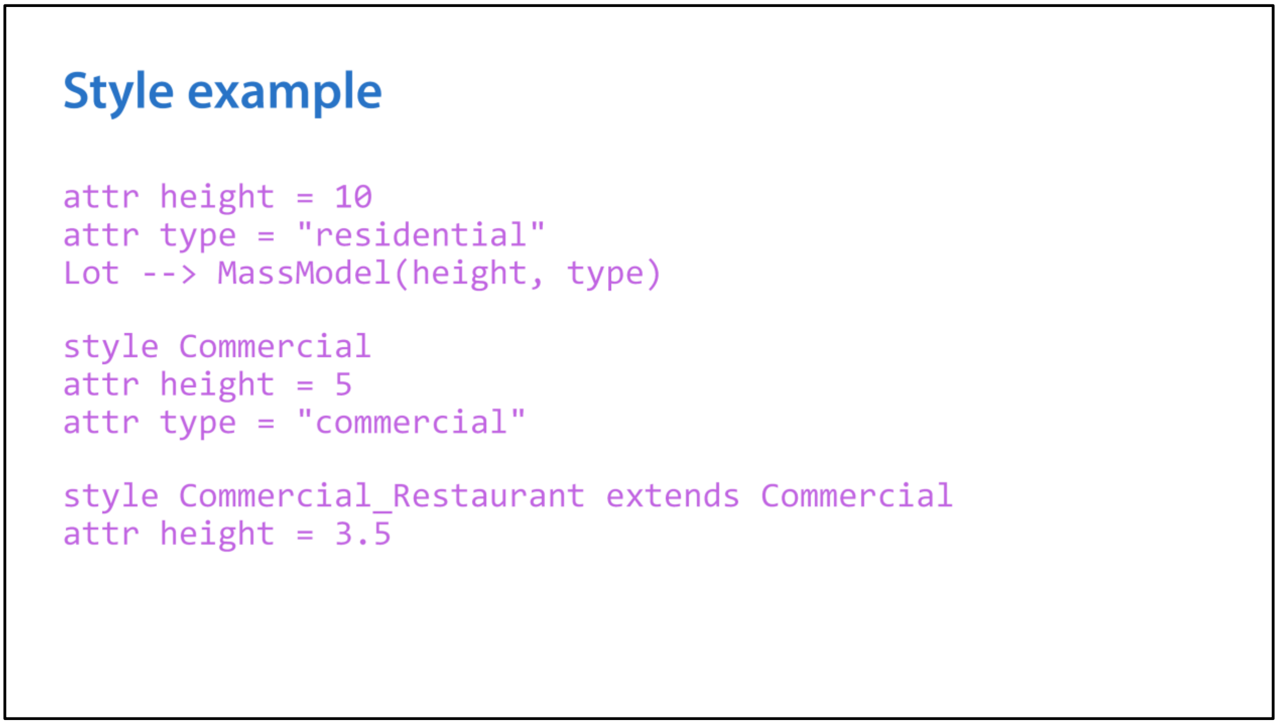
Images: Krecklau 2012



Styles

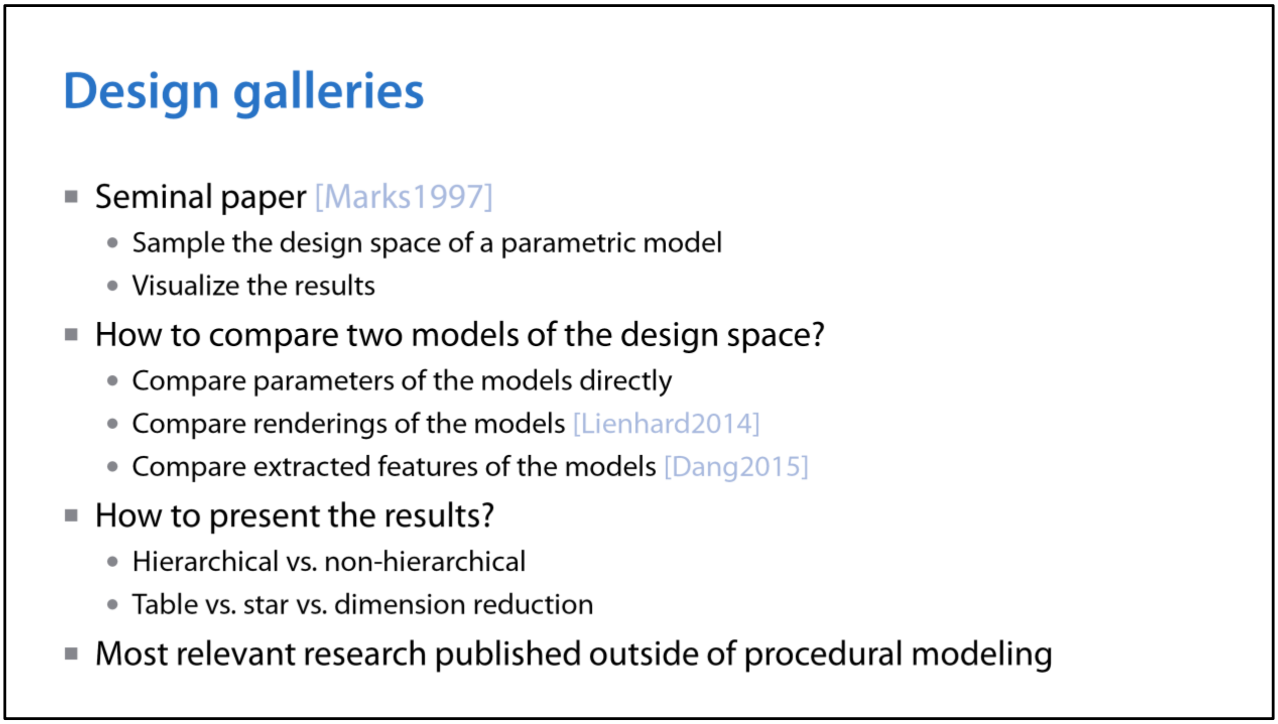
- Concept used in CityEngine
- Multiple styles in a rules file
- Styles define attributes and rules
- Styles can be derived from other styles
  - All attributes and rules are inherited from the parent style per default

Image: CityEngine 2015 Online Help System



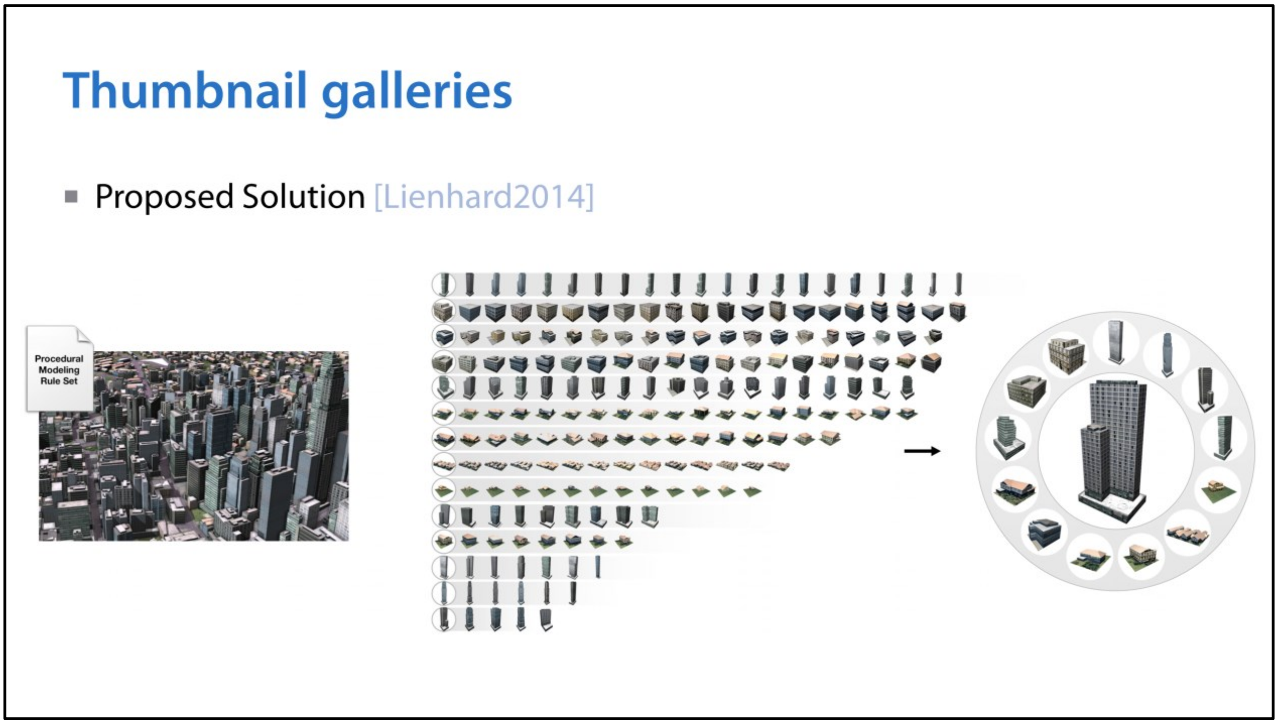
Style example

```
attr height = 10
attr type = "residential"
Lot --> MassModel(height, type)

style Commercial
attr height = 5
attr type = "commercial"

style Commercial_Restaurant extends Commercial
attr height = 3.5
```



Design galleries

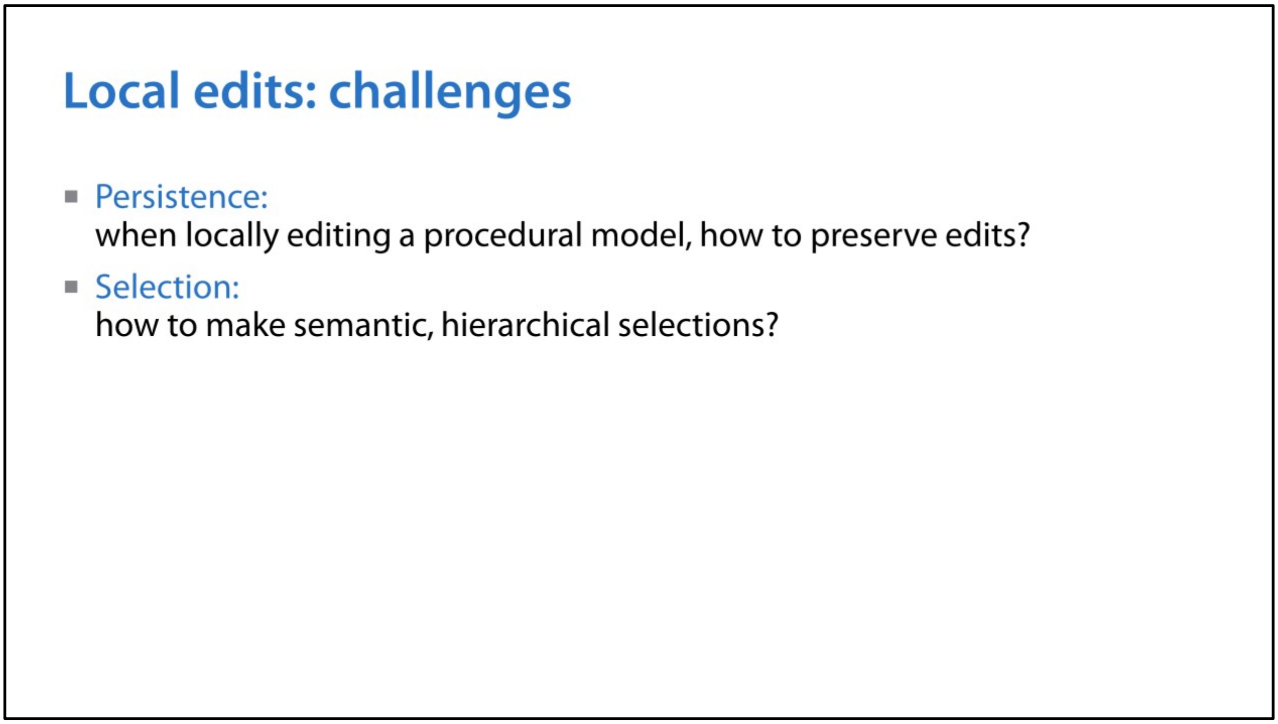
- Seminal paper [Marks1997]
  - Sample the design space of a parametric model
  - Visualize the results
- How to compare two models of the design space?
  - Compare parameters of the models directly
  - Compare renderings of the models [Lienhard2014]
  - Compare extracted features of the models [Dang2015]
- How to present the results?
  - Hierarchical vs. non-hierarchical
  - Table vs. star vs. dimension reduction
- Most relevant research published outside of procedural modeling



Thumbnail galleries

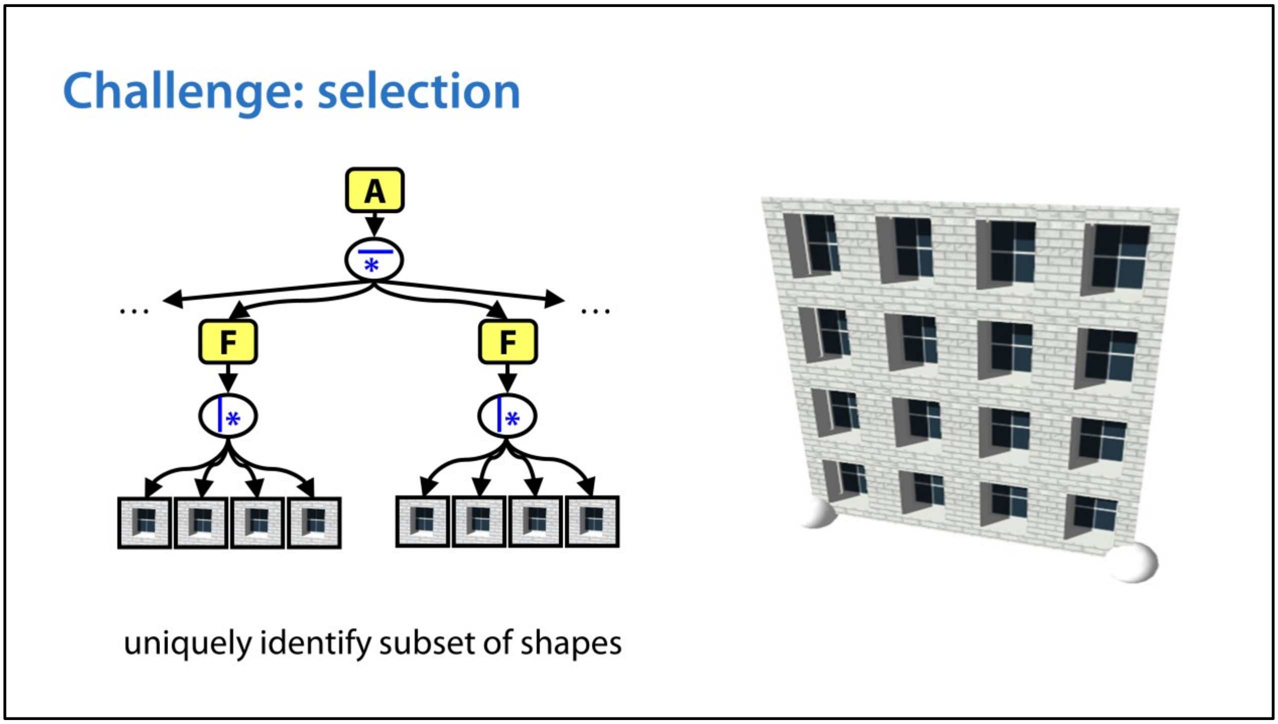
- Proposed Solution [Lienhard2014]

Image: Lienhard 2014



Local edits: challenges

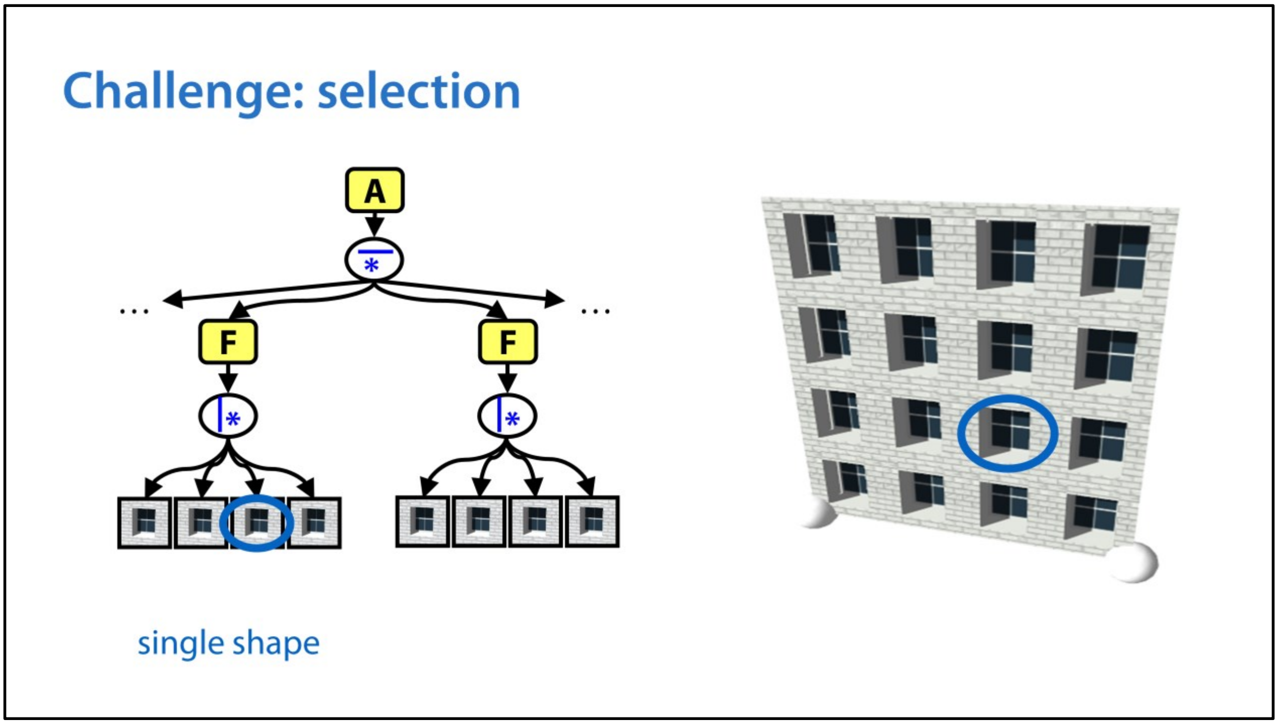
- Persistence:
  when locally editing a procedural model, how to preserve edits?
- Selection:
  how to make semantic, hierarchical selections?



Challenge: selection

uniquely identify subset of shapes

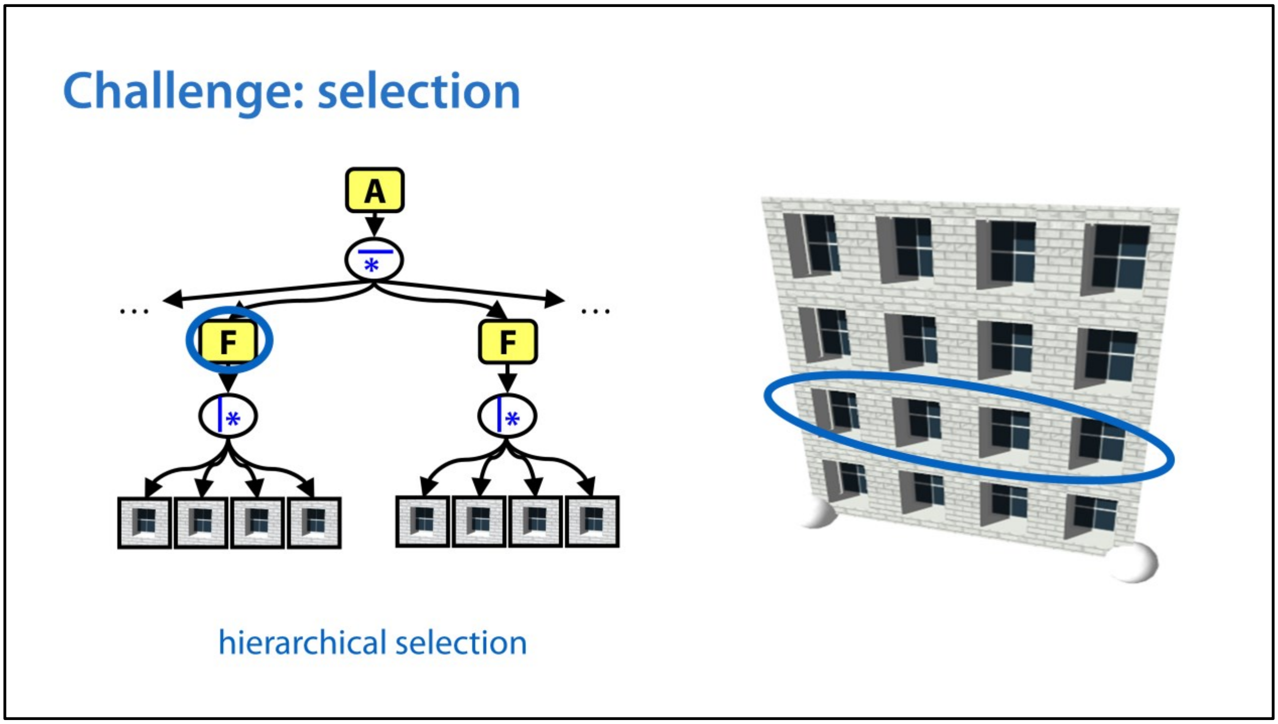
Images: Lipp 2008



Challenge: selection

single shape

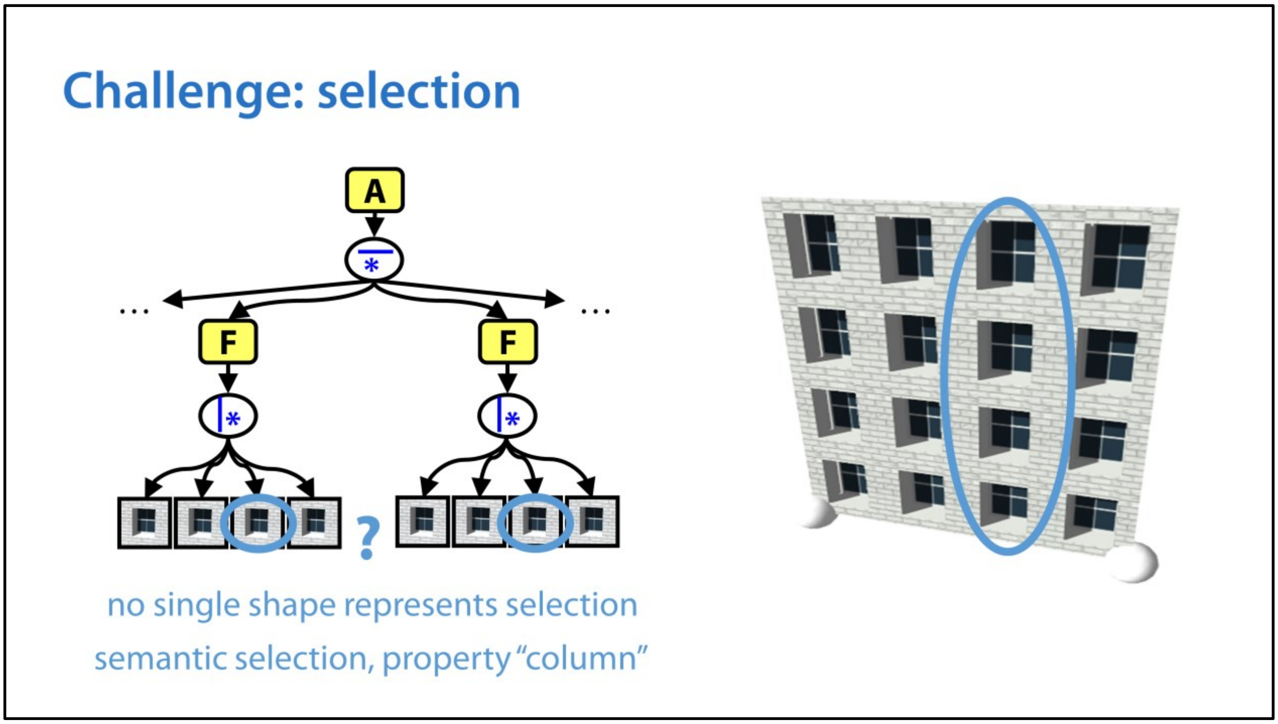
Images: Lipp 2008



Challenge: selection

hierarchical selection

Images: Lipp 2008



Challenge: selection

no single shape represents selection

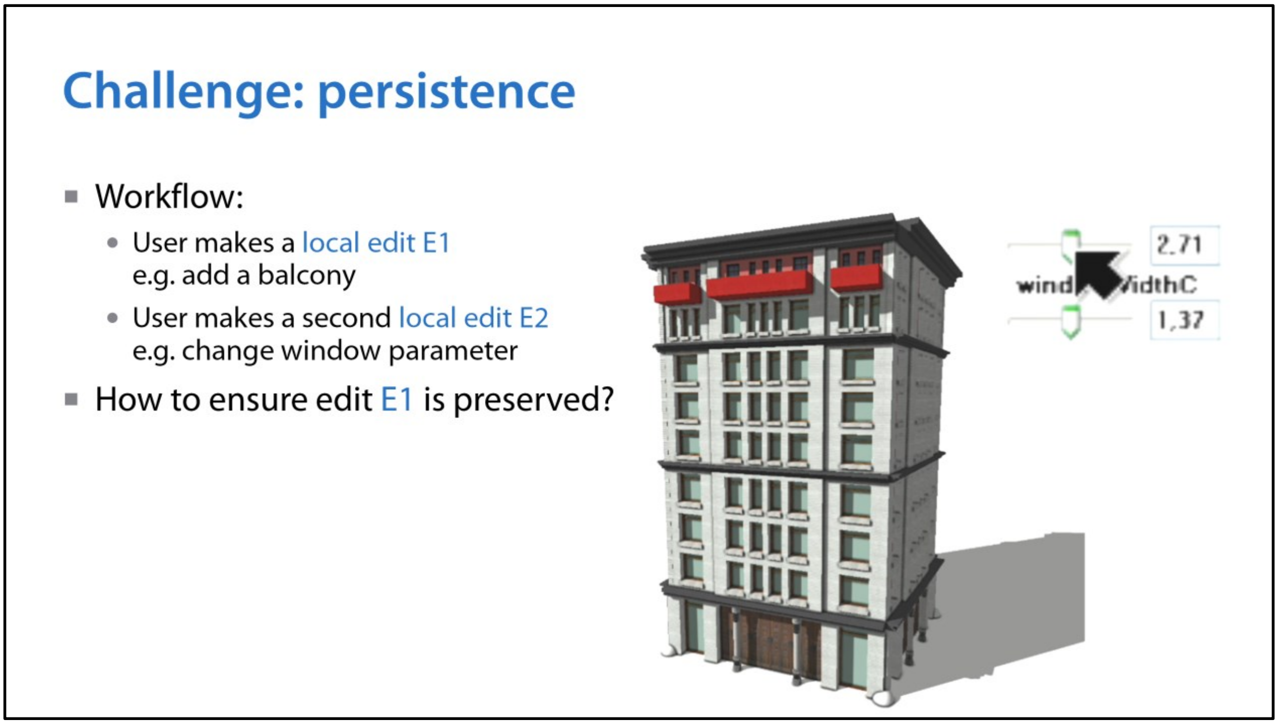
semantic selection, property "column"

Images: Lipp 2008

Challenge: persistence

- Workflow:
  - User makes a local edit E1 e.g. add a balcony

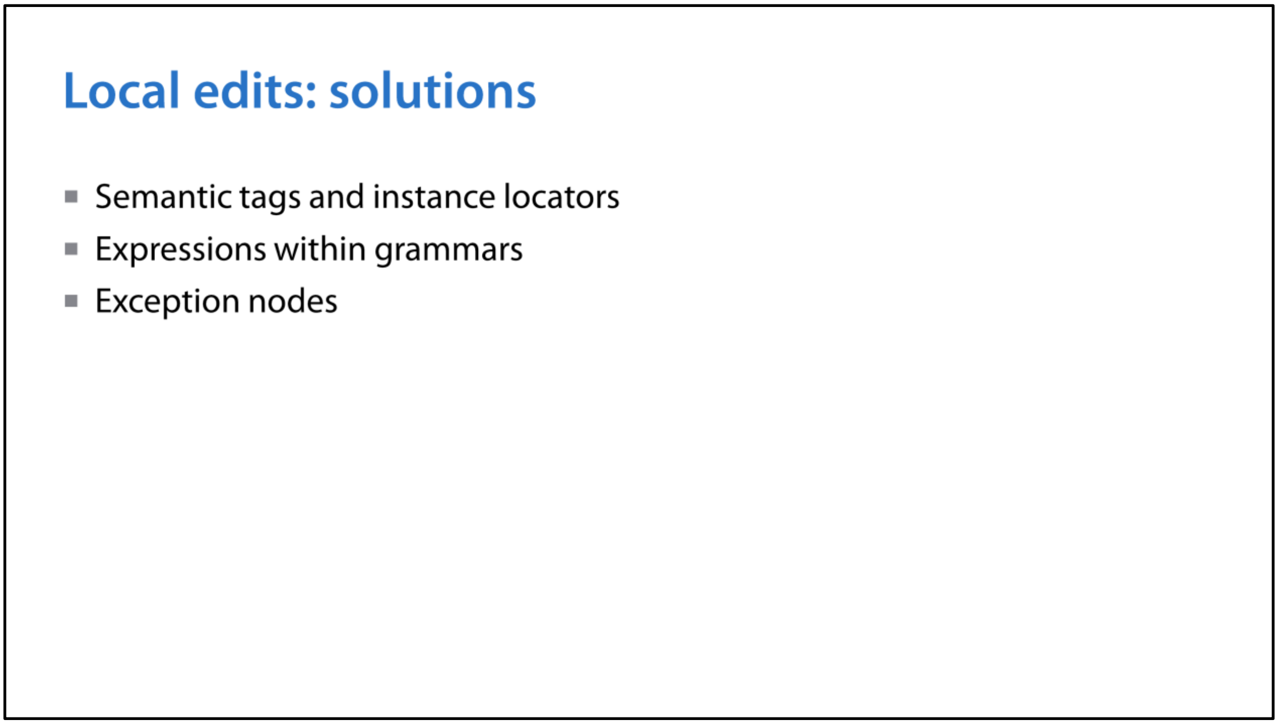
Video: Lipp 2008



Challenge: persistence

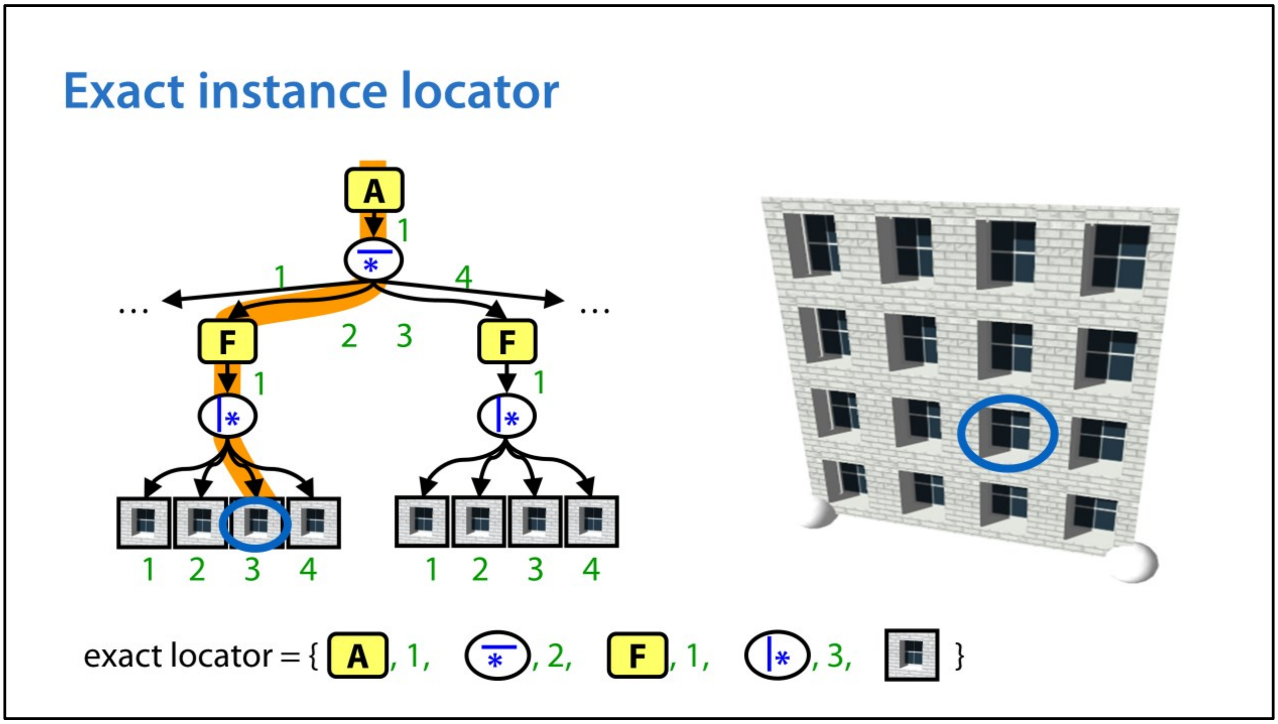
- Workflow:
  - User makes a local edit E1
    e.g. add a balcony
  - User makes a second local edit E2
    e.g. change window parameter
- How to ensure edit E1 is preserved?

Video: Lipp 2008



Local edits: solutions

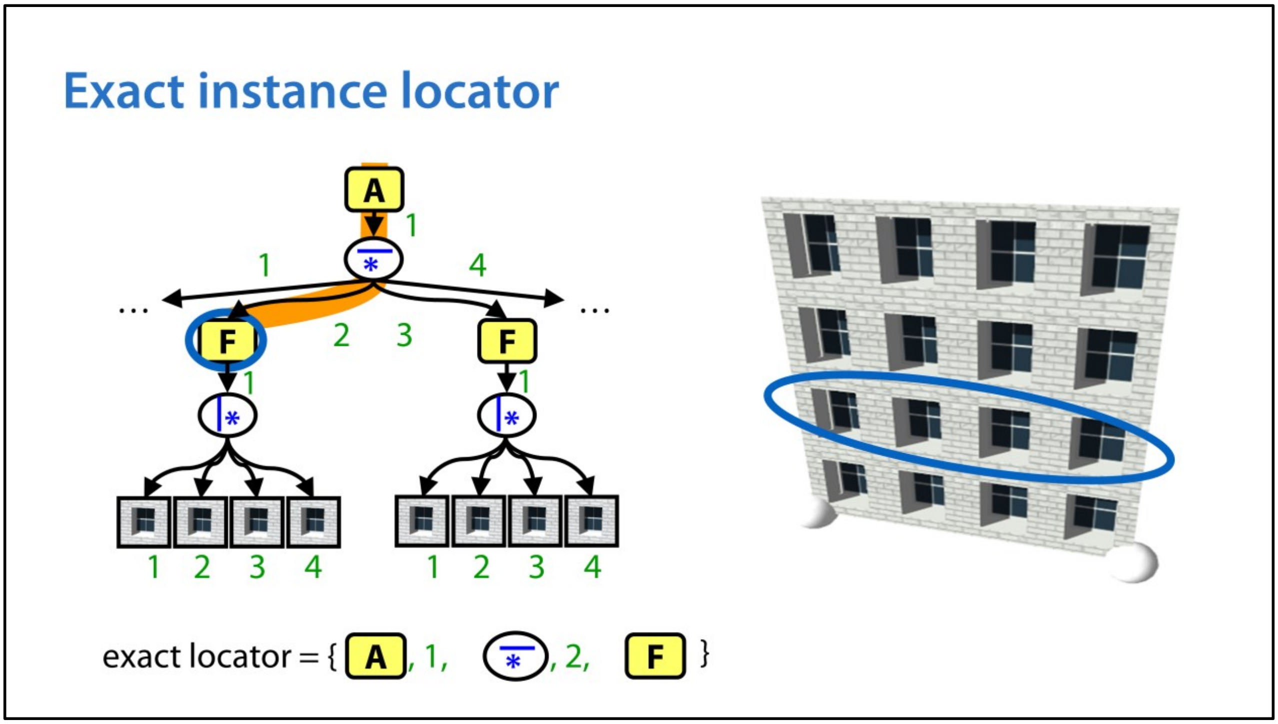
- Semantic tags and instance locators
- Expressions within grammars
- Exception nodes



Exact instance locator

exact locator = { A, 1, *, 2, F, 1, |*, 3, ▣ }

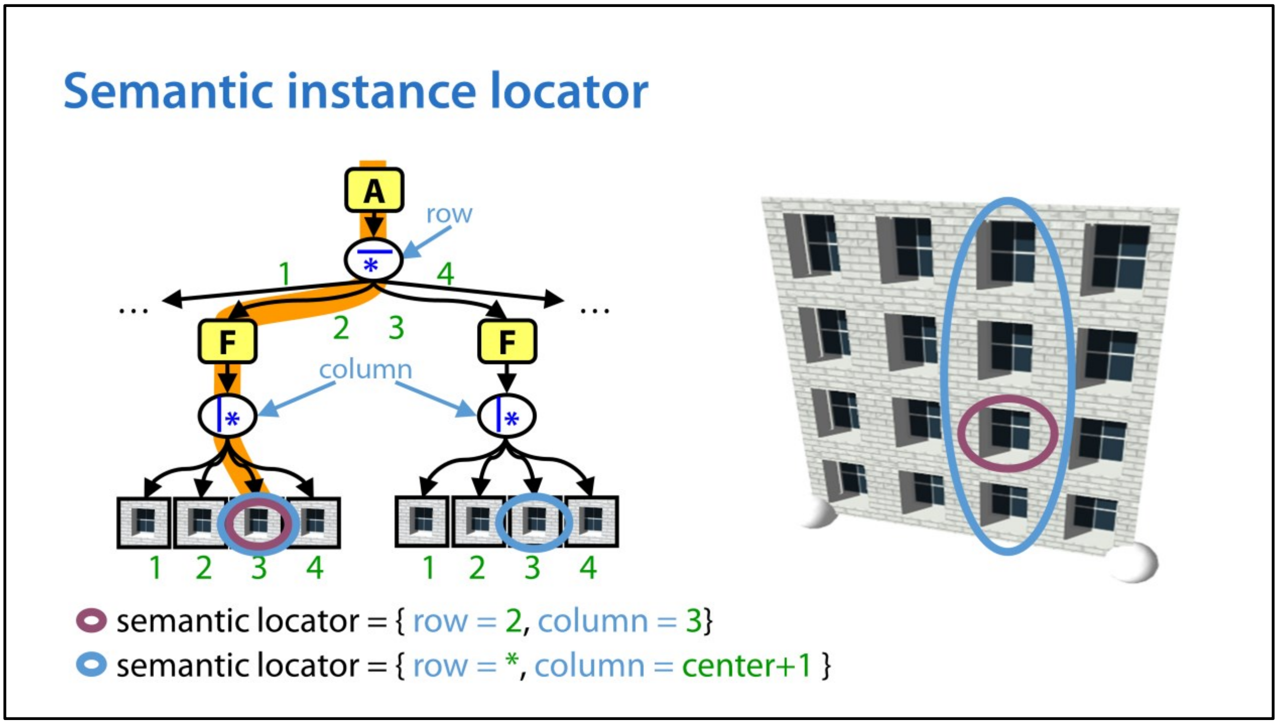
Image: Lipp 2008



Exact instance locator

exact locator = { A, 1, *, 2, F }

Image: Lipp 2008

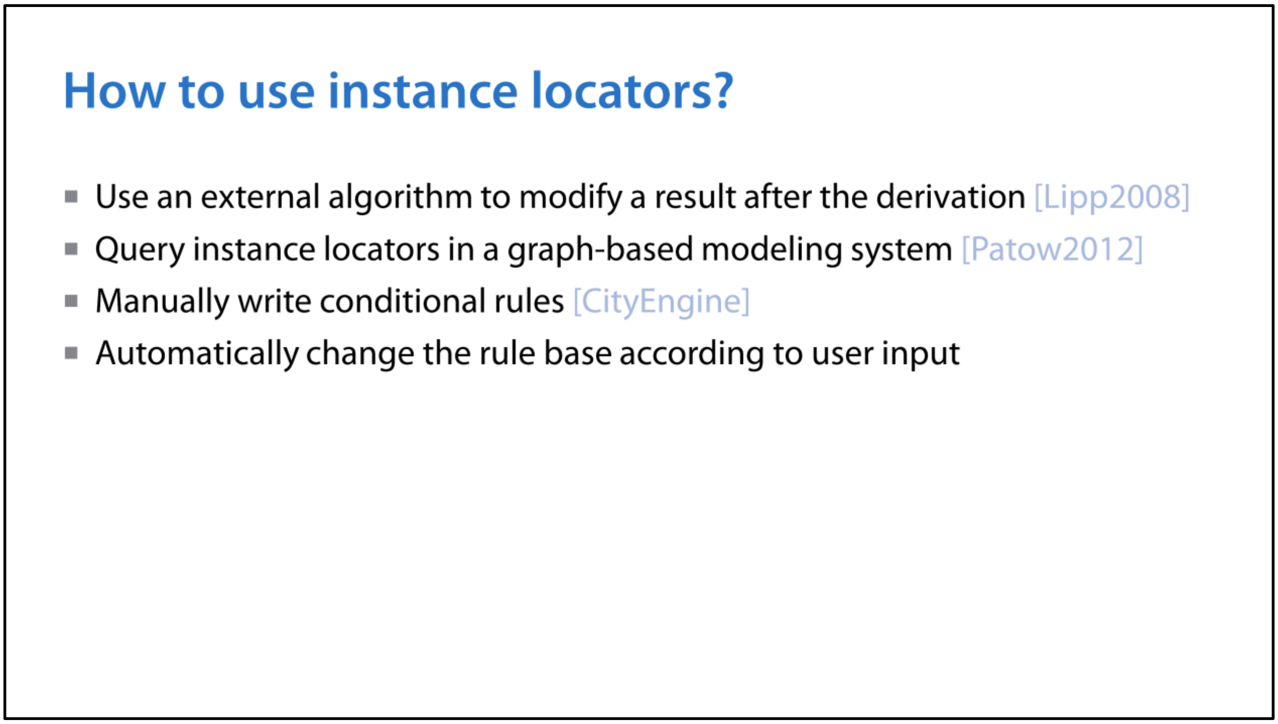


Semantic instance locator

○ semantic locator = { row = 2, column = 3}

○ semantic locator = { row = *, column = center+1 }

Image: Lipp 2008



How to use instance locators?

- Use an external algorithm to modify a result after the derivation [Lipp2008]
- Query instance locators in a graph-based modeling system [Patow2012]
- Manually write conditional rules [CityEngine]
- Automatically change the rule base according to user input

Expression within grammars

- Manually extend a grammar using instance locators
- E.g., CityEngine operation `getTreeKey`
- Use `getTreeKey` to write conditional rules
- `getTreeKey` returns a sequence of numbers
- Marked floor would have a key 0 – 0 - 1

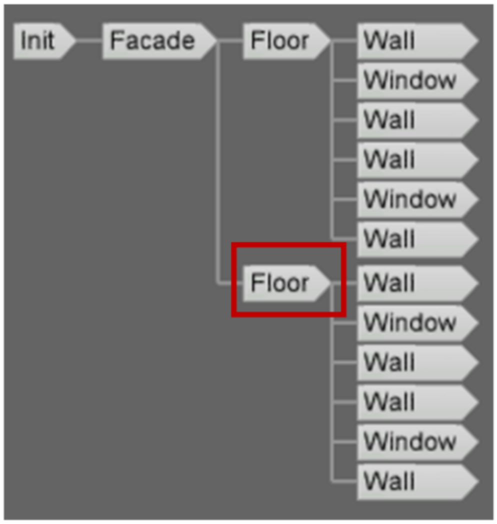
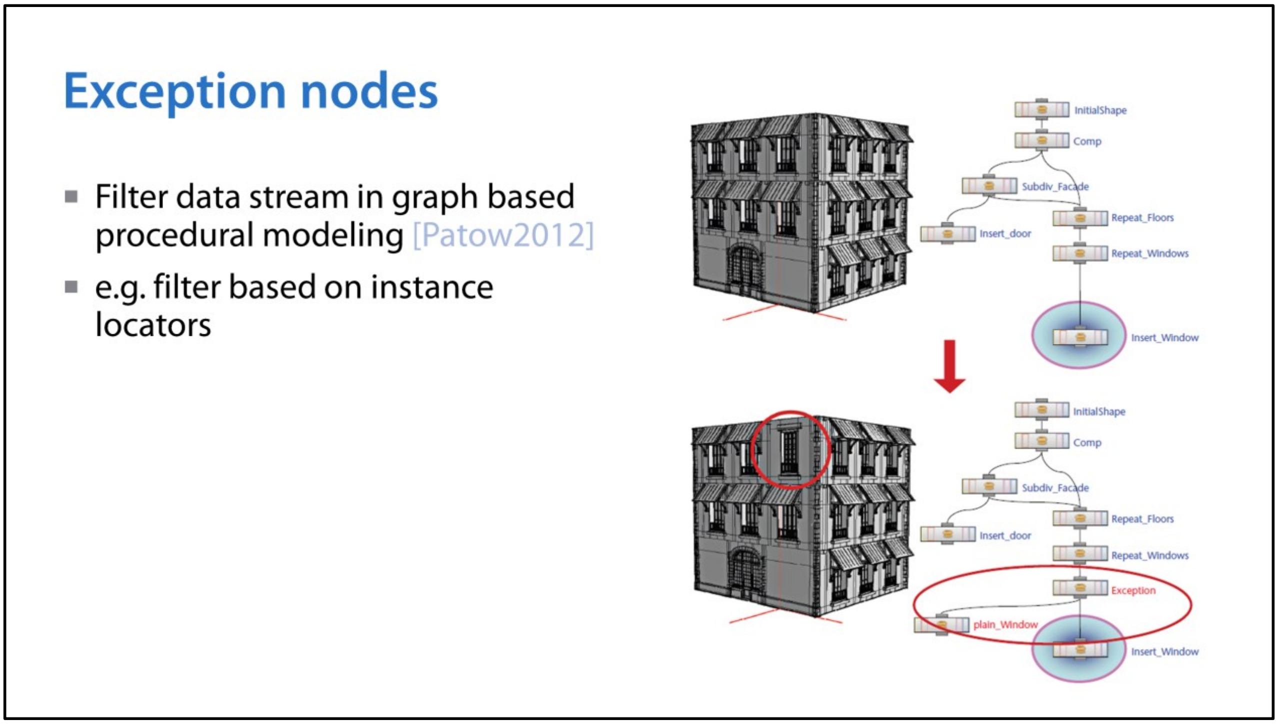


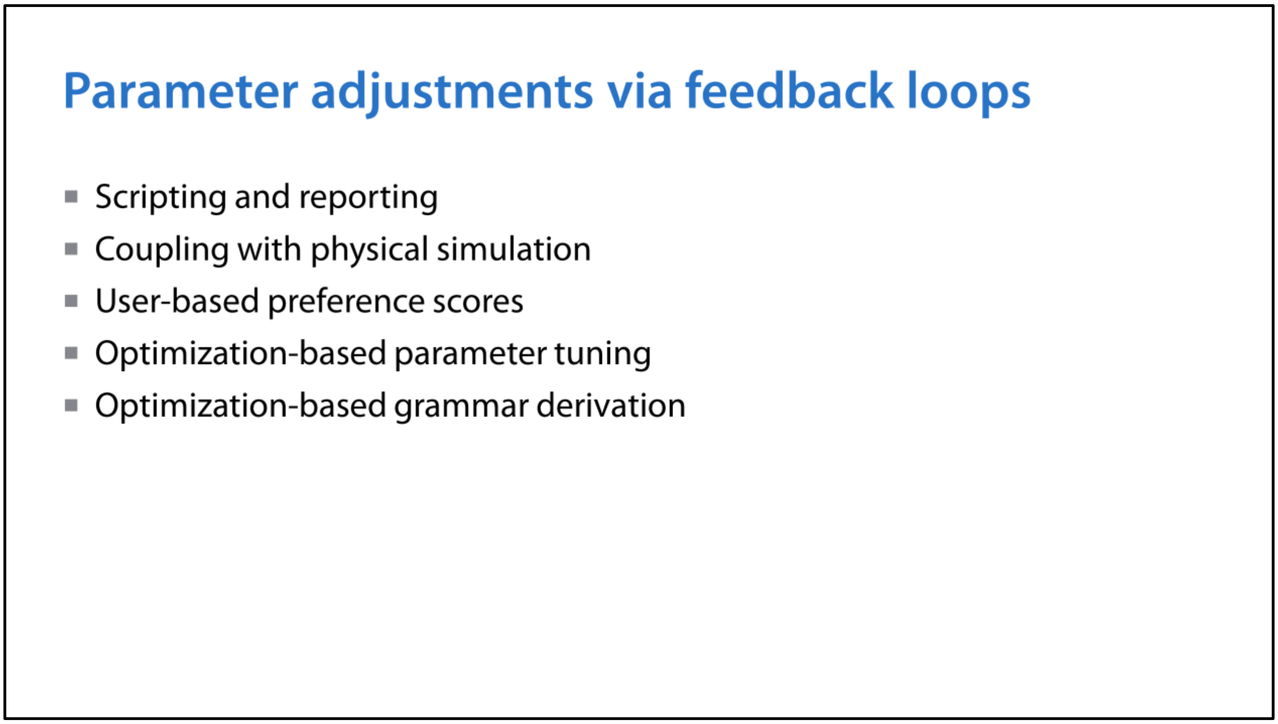
Image: CityEngine online help



Exception nodes

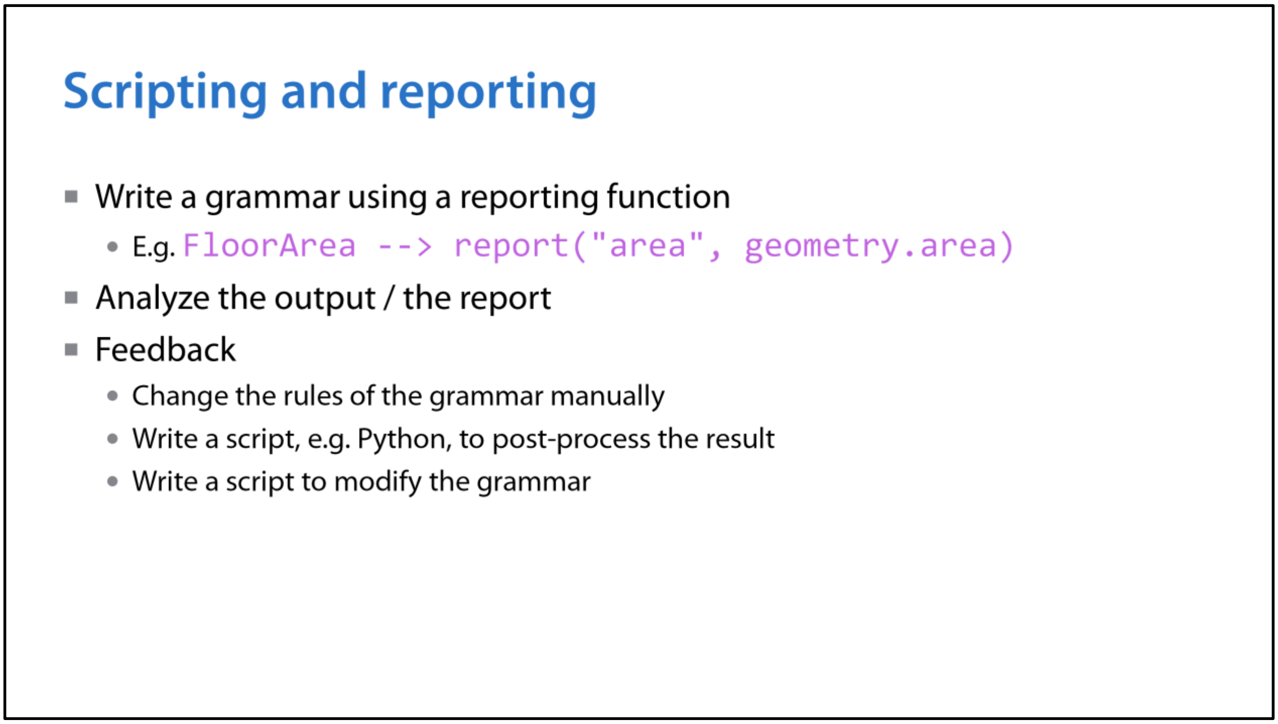
- Filter data stream in graph based procedural modeling [Patow2012]
- e.g. filter based on instance locators

Image: Patow 2012



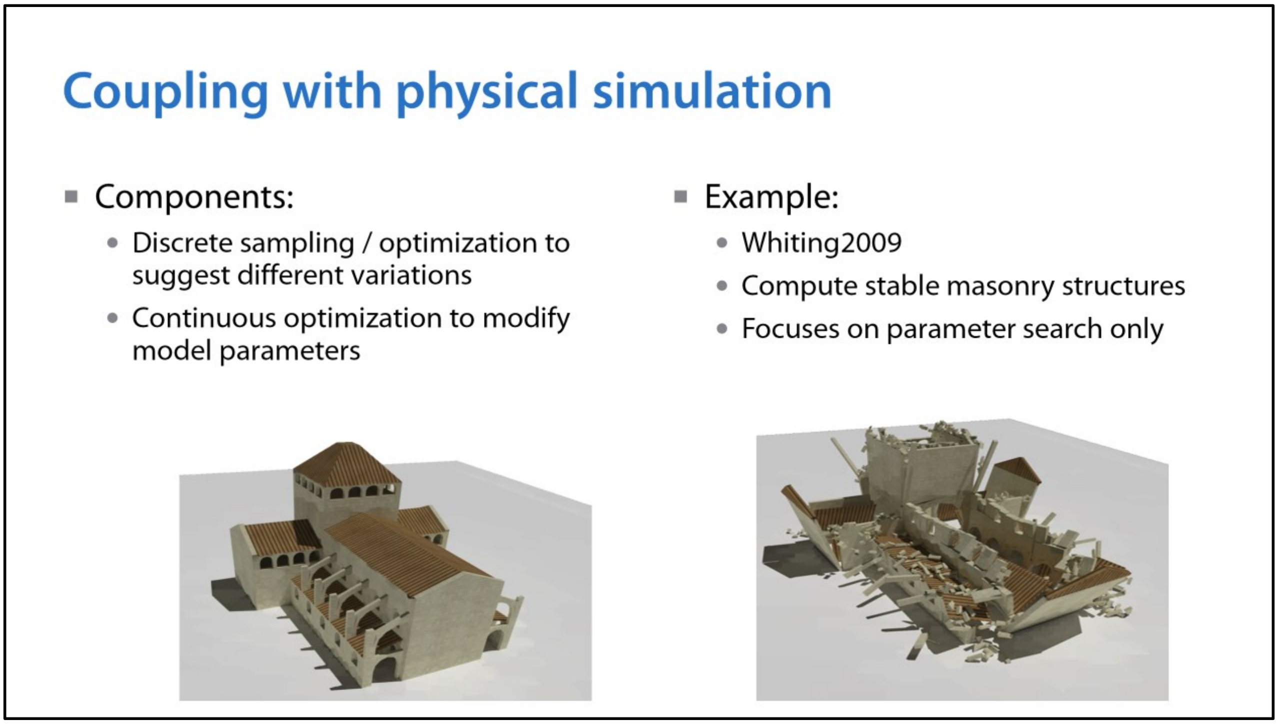
Parameter adjustments via feedback loops

- Scripting and reporting
- Coupling with physical simulation
- User-based preference scores
- Optimization-based parameter tuning
- Optimization-based grammar derivation



Scripting and reporting

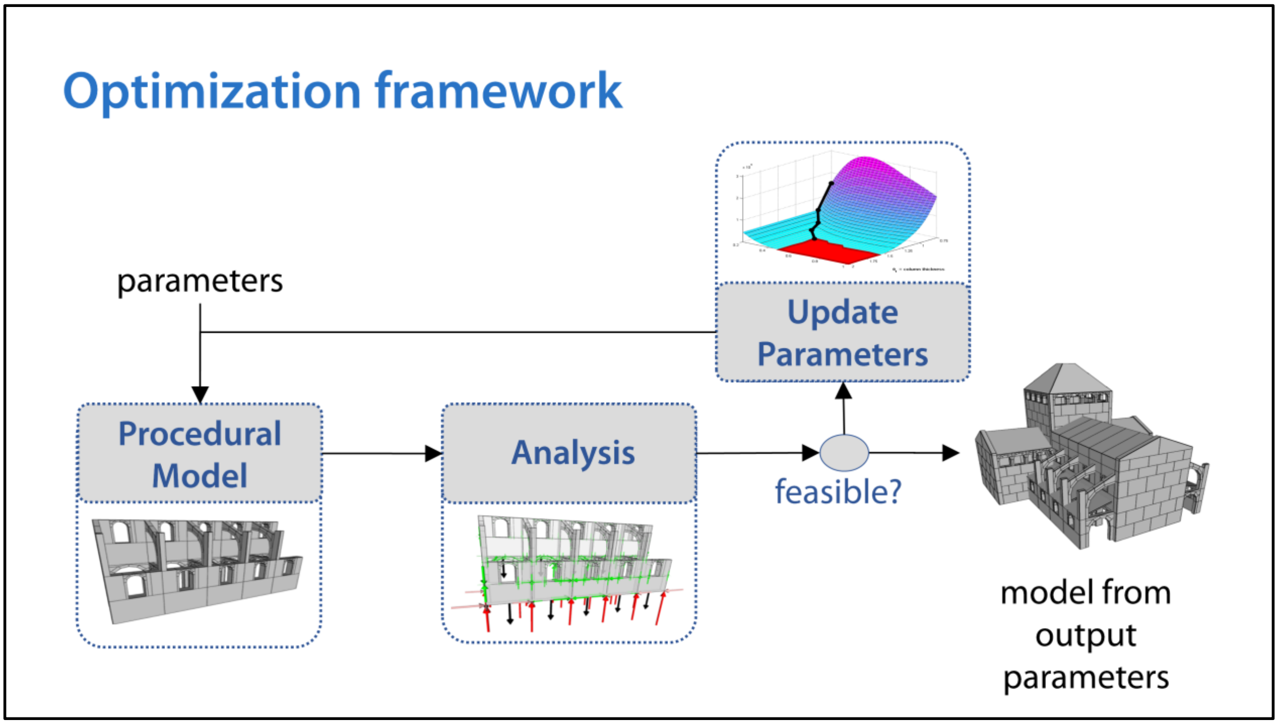
- Write a grammar using a reporting function
  - E.g. `FloorArea --> report("area", geometry.area)`
- Analyze the output / the report
- Feedback
  - Change the rules of the grammar manually
  - Write a script, e.g. Python, to post-process the result
  - Write a script to modify the grammar



Coupling with physical simulation

- Components:
  - Discrete sampling / optimization to suggest different variations
  - Continuous optimization to modify model parameters
- Example:
  - Whiting2009
  - Compute stable masonry structures
  - Focuses on parameter search only

Image: Whiting 2009



Optimization framework

parameters

Update Parameters

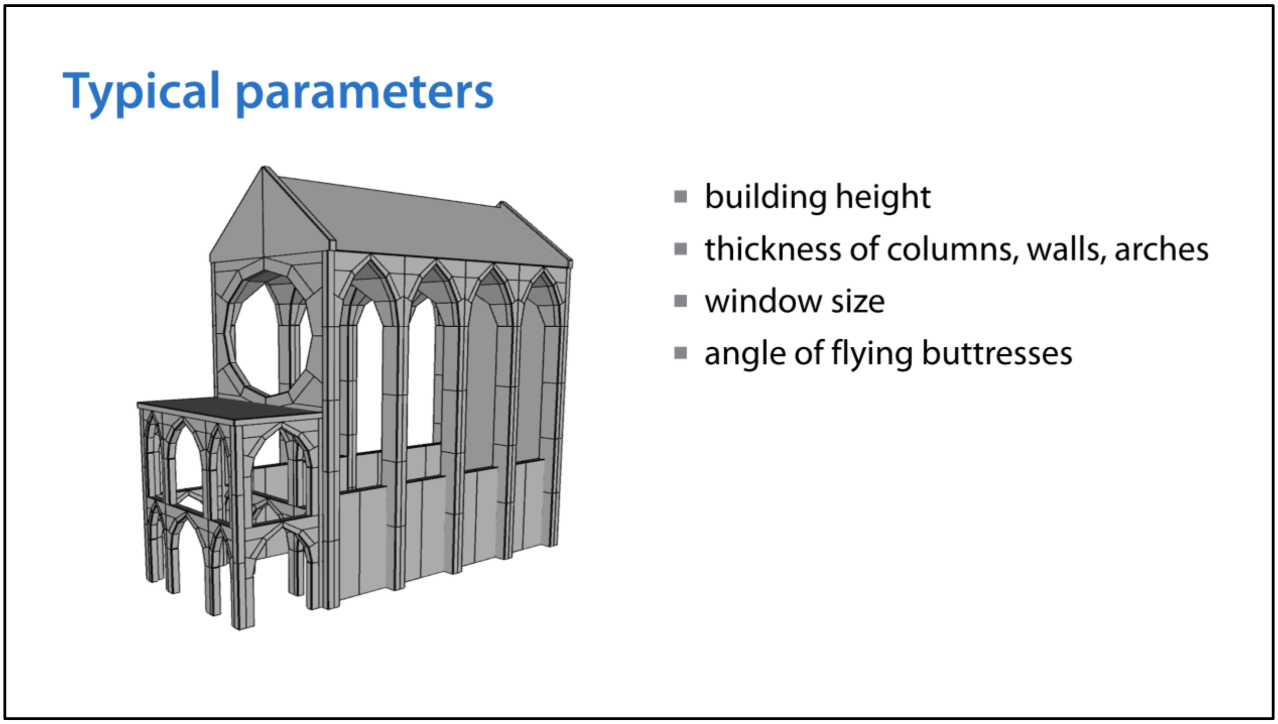
Procedural Model

Analysis

feasible?

model from output parameters

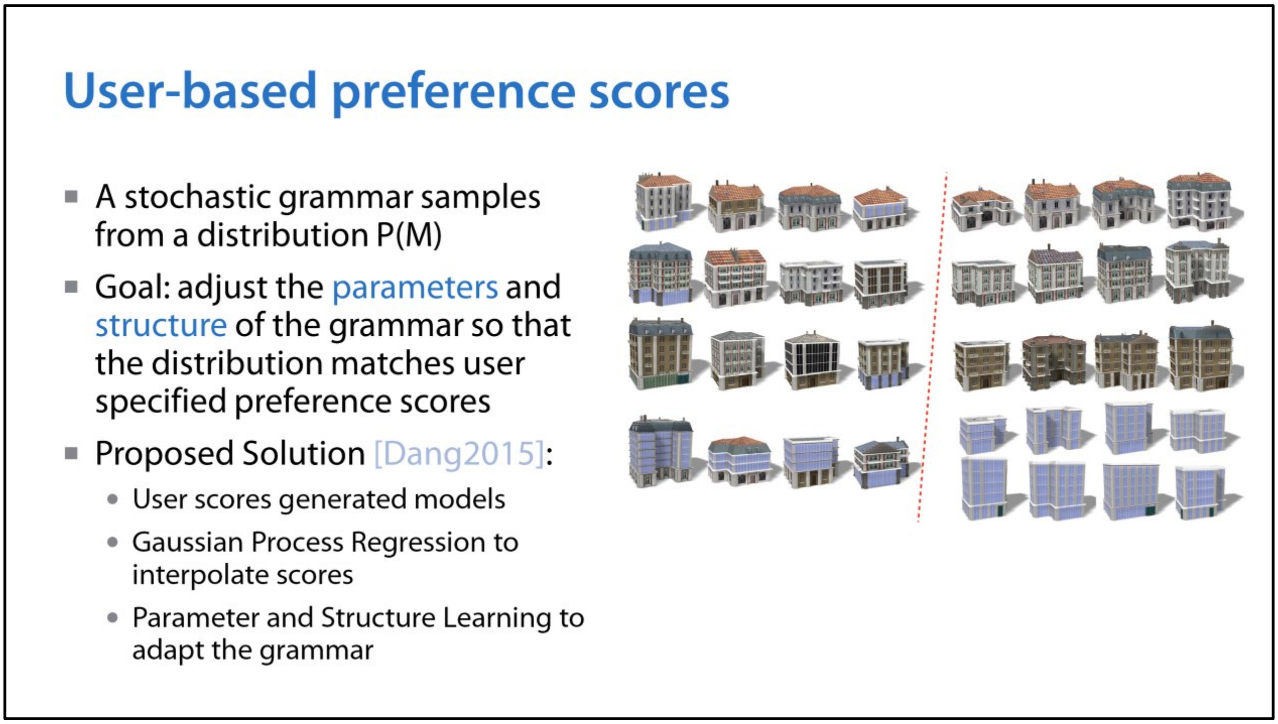
Images: Whiting 2009



Typical parameters

- building height
- thickness of columns, walls, arches
- window size
- angle of flying buttresses

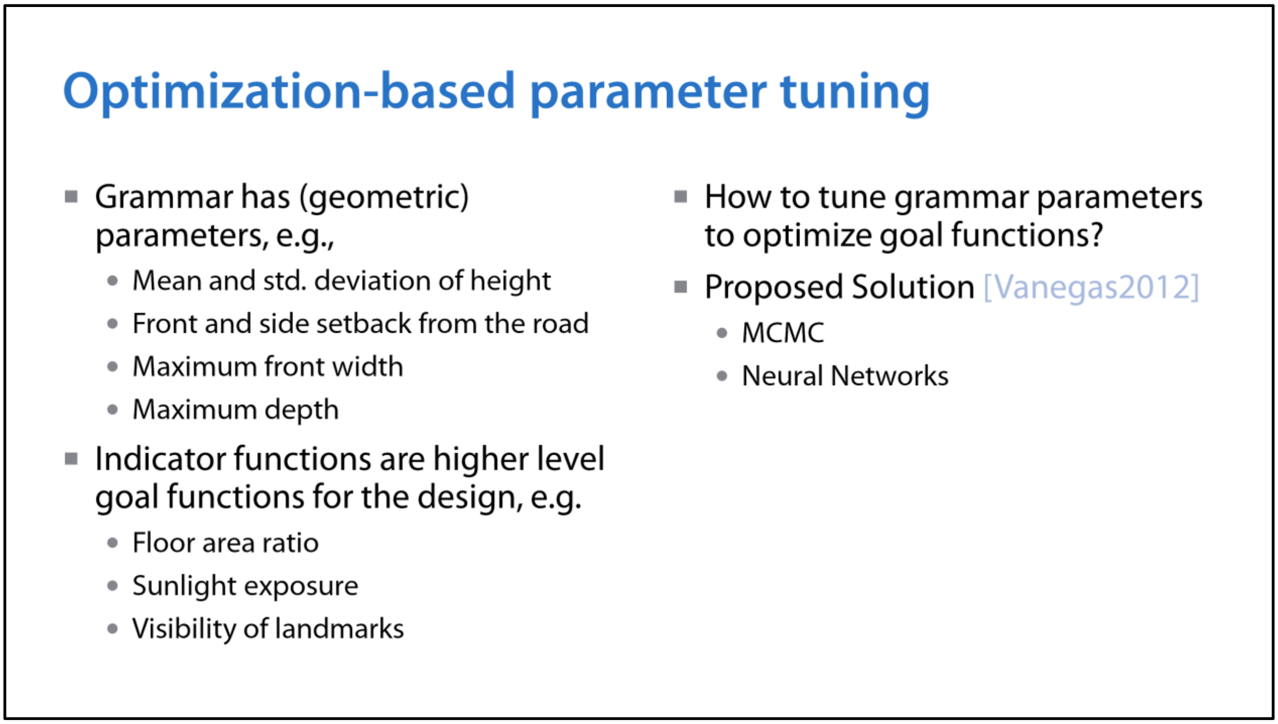
Image: Whiting 2009



User-based preference scores

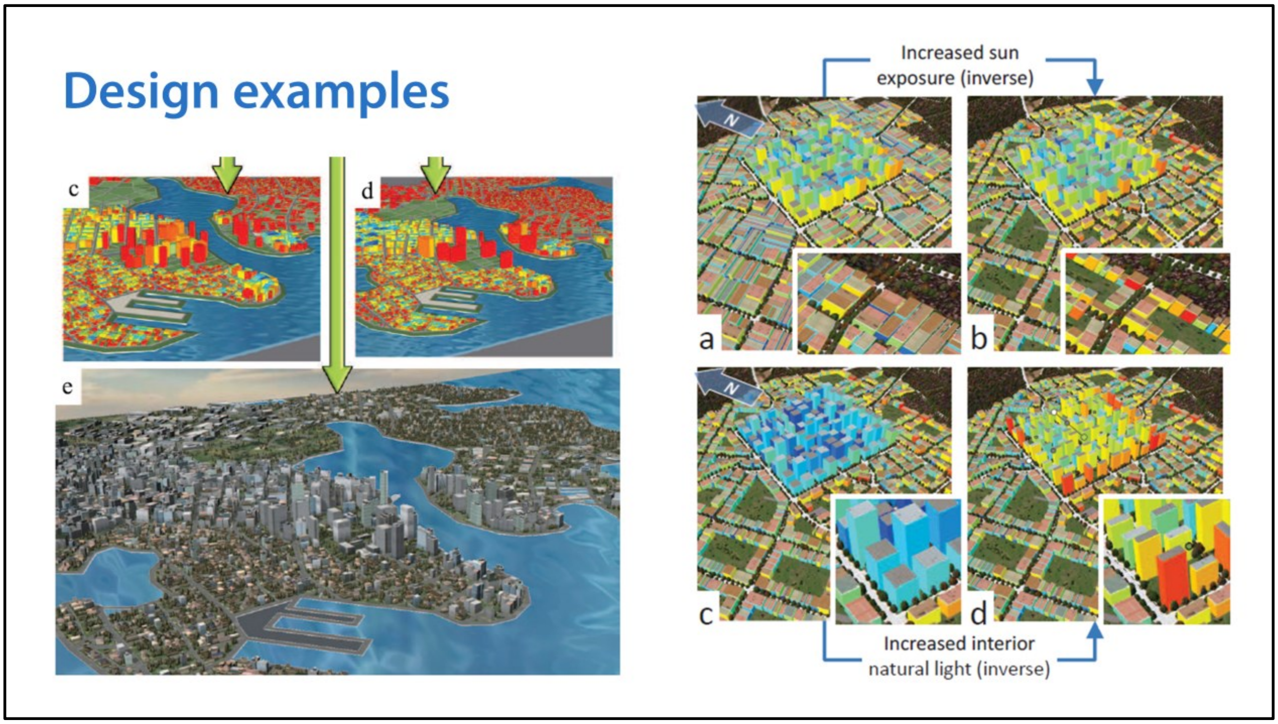
- A stochastic grammar samples from a distribution P(M)
- Goal: adjust the parameters and structure of the grammar so that the distribution matches user specified preference scores
- Proposed Solution [Dang2015]:
  - User scores generated models
  - Gaussian Process Regression to interpolate scores
  - Parameter and Structure Learning to adapt the grammar

Image: Dang 2015



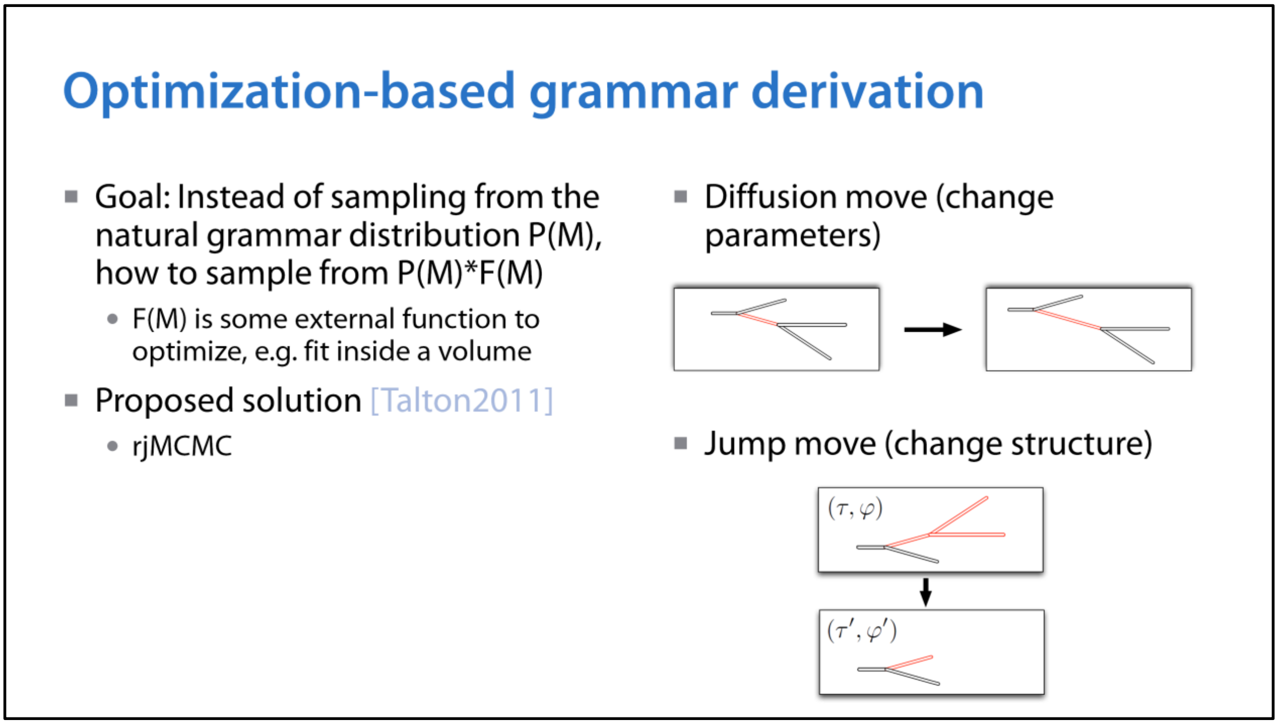
Optimization-based parameter tuning

- Grammar has (geometric) parameters, e.g.,
  - Mean and std. deviation of height
  - Front and side setback from the road
  - Maximum front width
  - Maximum depth
- Indicator functions are higher level goal functions for the design, e.g.
  - Floor area ratio
  - Sunlight exposure
  - Visibility of landmarks
- How to tune grammar parameters to optimize goal functions?
- Proposed Solution [Vanegas2012]
  - MCMC
  - Neural Networks



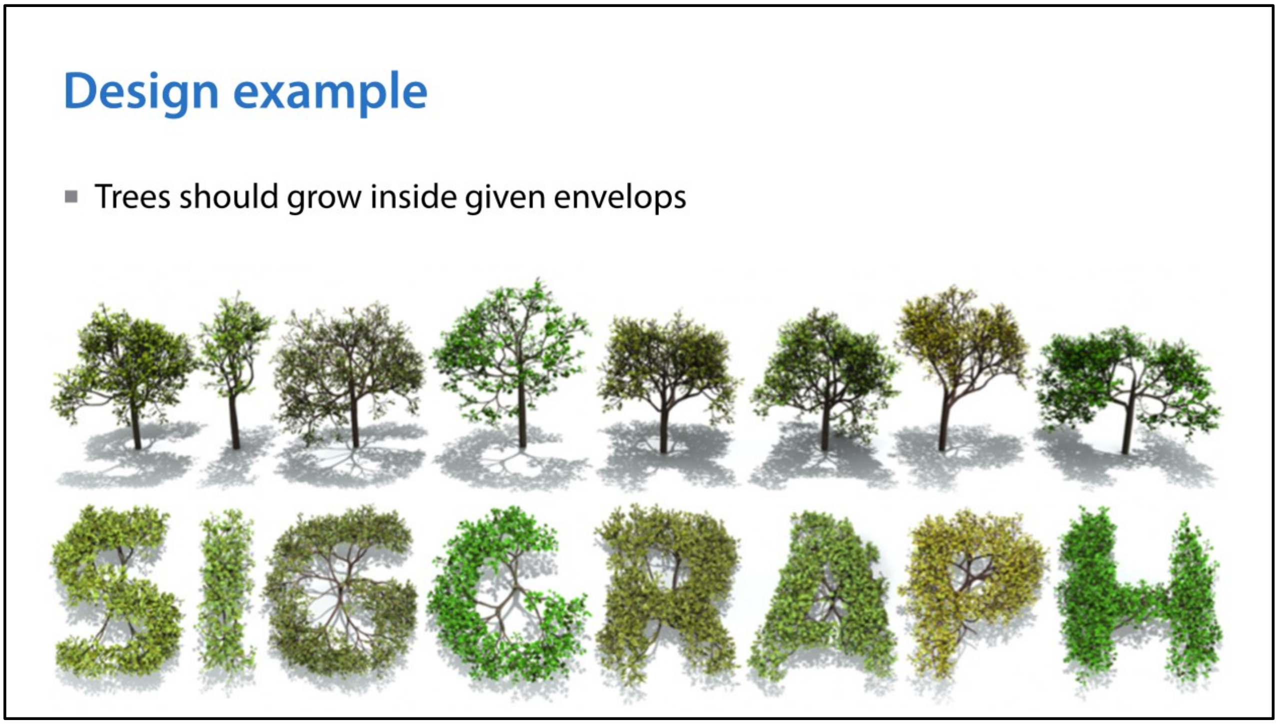
Design examples

Images: Vanegas2012



Optimization-based grammar derivation

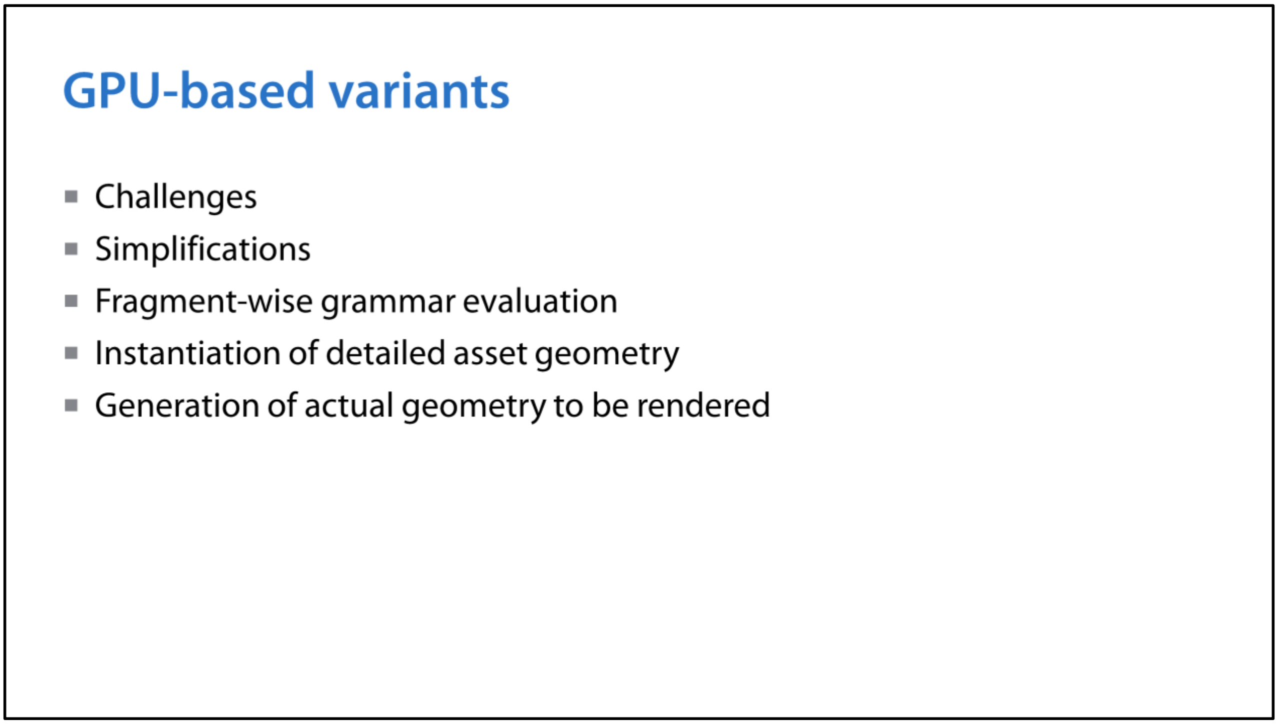
- Goal: Instead of sampling from the natural grammar distribution P(M), how to sample from P(M)*F(M)
  - F(M) is some external function to optimize, e.g. fit inside a volume
- Proposed solution [Talton2011]
  - rjMCMC
- Diffusion move (change parameters)
- Jump move (change structure)



Design example

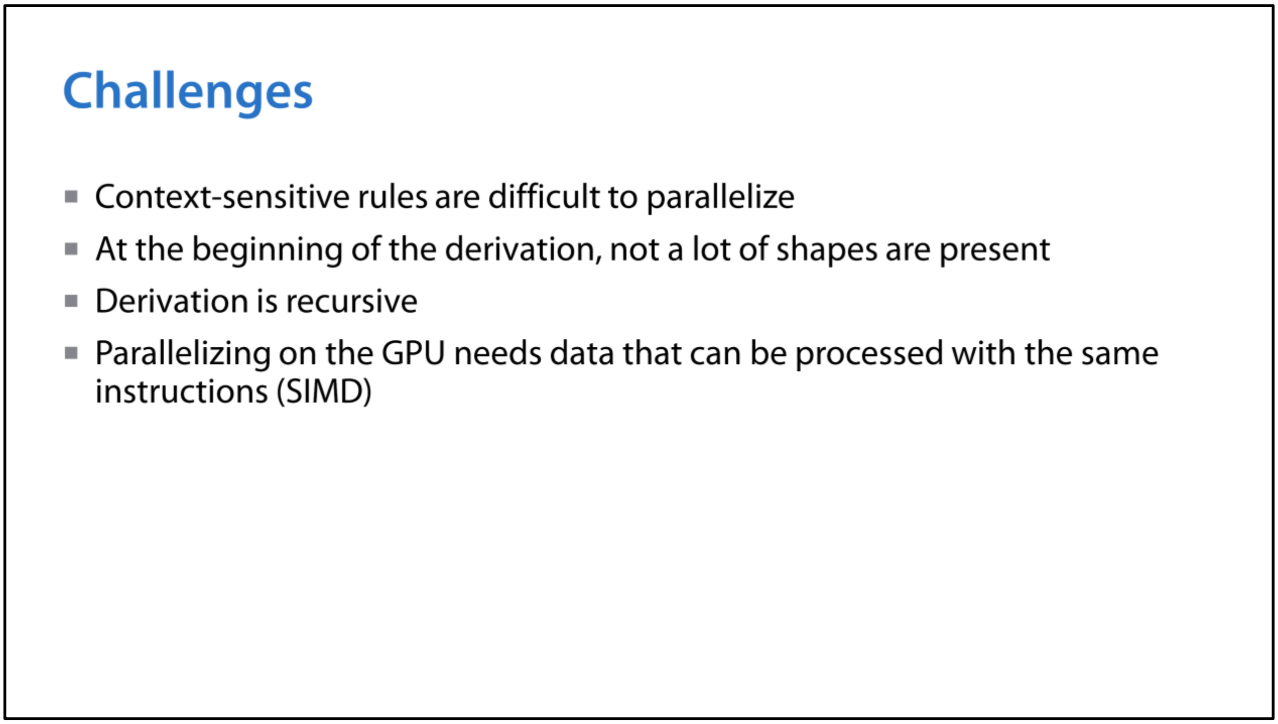
- Trees should grow inside given envelops

Image: Talton2011



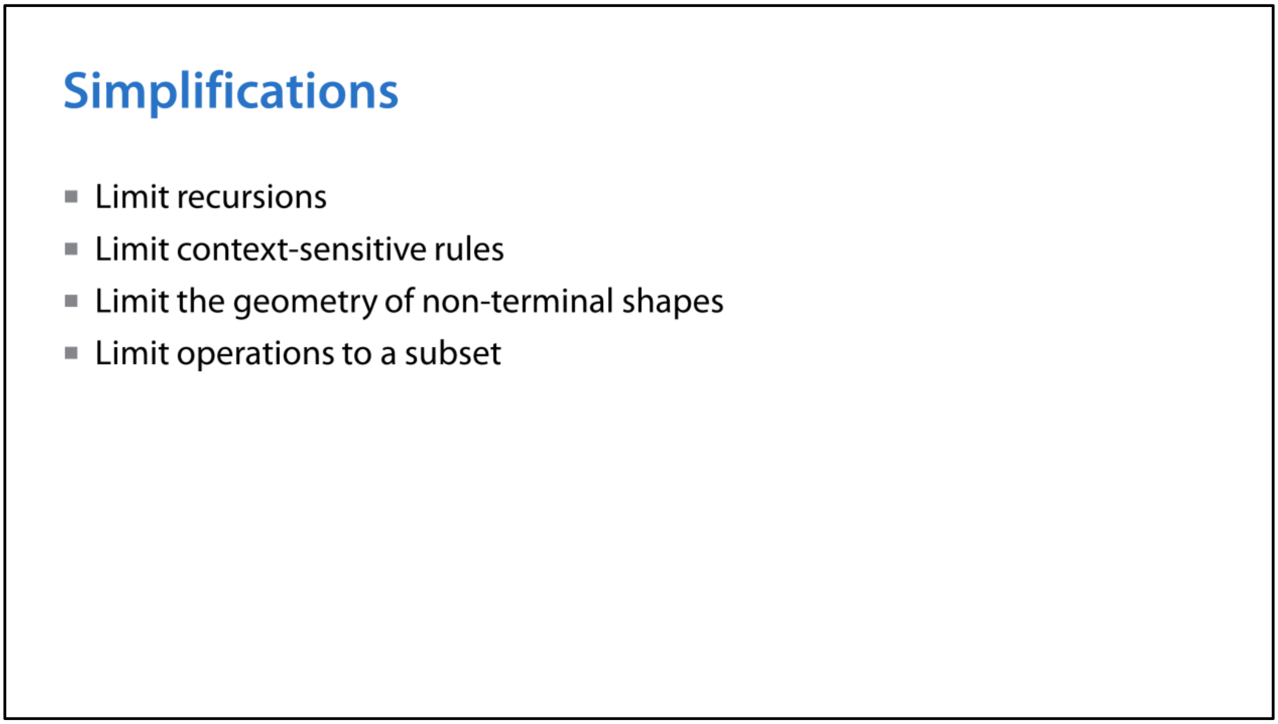
GPU-based variants

- Challenges
- Simplifications
- Fragment-wise grammar evaluation
- Instantiation of detailed asset geometry
- Generation of actual geometry to be rendered



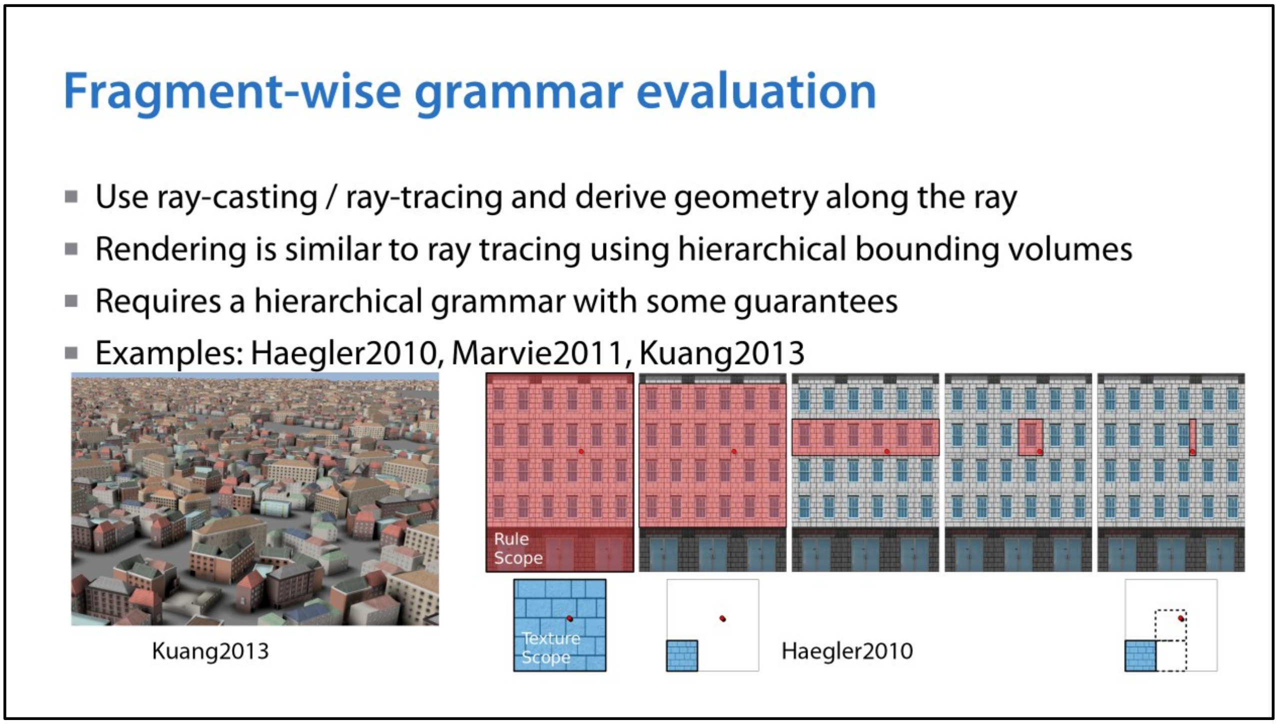
Challenges

- Context-sensitive rules are difficult to parallelize
- At the beginning of the derivation, not a lot of shapes are present
- Derivation is recursive
- Parallelizing on the GPU needs data that can be processed with the same instructions (SIMD)



Simplifications

- Limit recursions
- Limit context-sensitive rules
- Limit the geometry of non-terminal shapes
- Limit operations to a subset

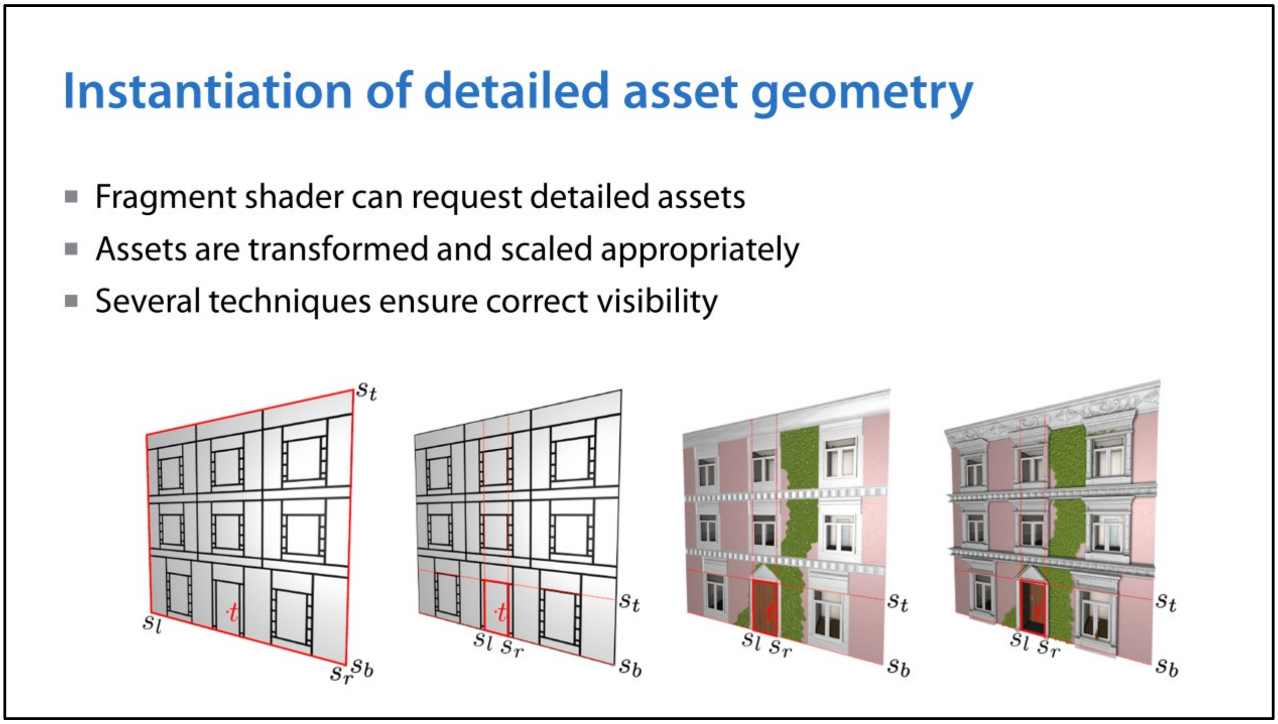


Fragment-wise grammar evaluation

- Use ray-casting / ray-tracing and derive geometry along the ray
- Rendering is similar to ray tracing using hierarchical bounding volumes
- Requires a hierarchical grammar with some guarantees
- Examples: Haegler2010, Marvie2011, Kuang2013

Kuang2013

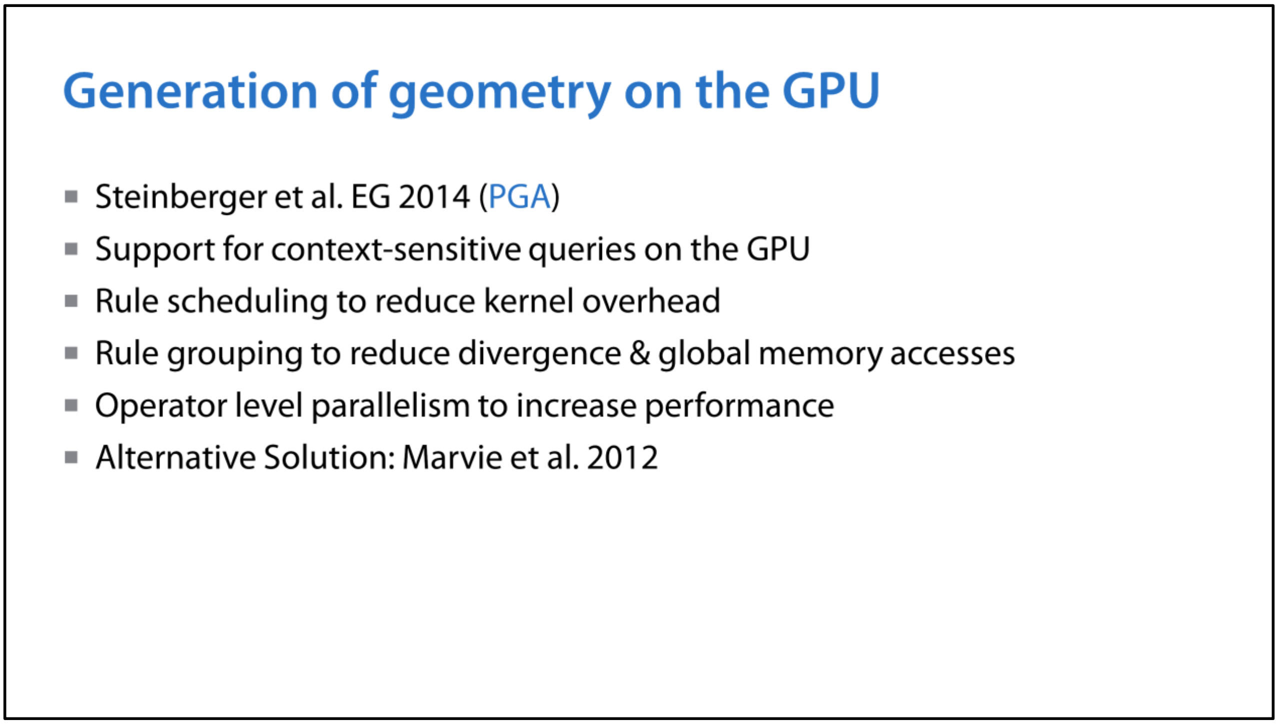
Haegler2010



Instantiation of detailed asset geometry

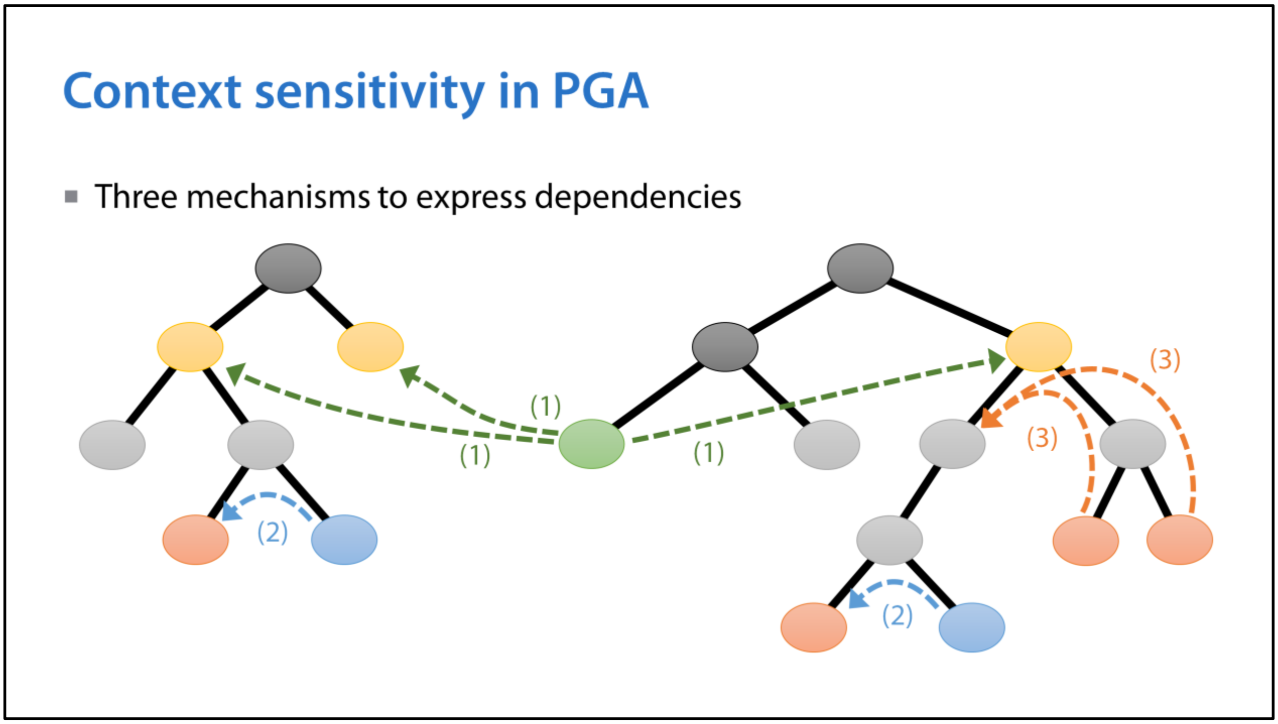
- Fragment shader can request detailed assets
- Assets are transformed and scaled appropriately
- Several techniques ensure correct visibility

Image: Krecklau2013



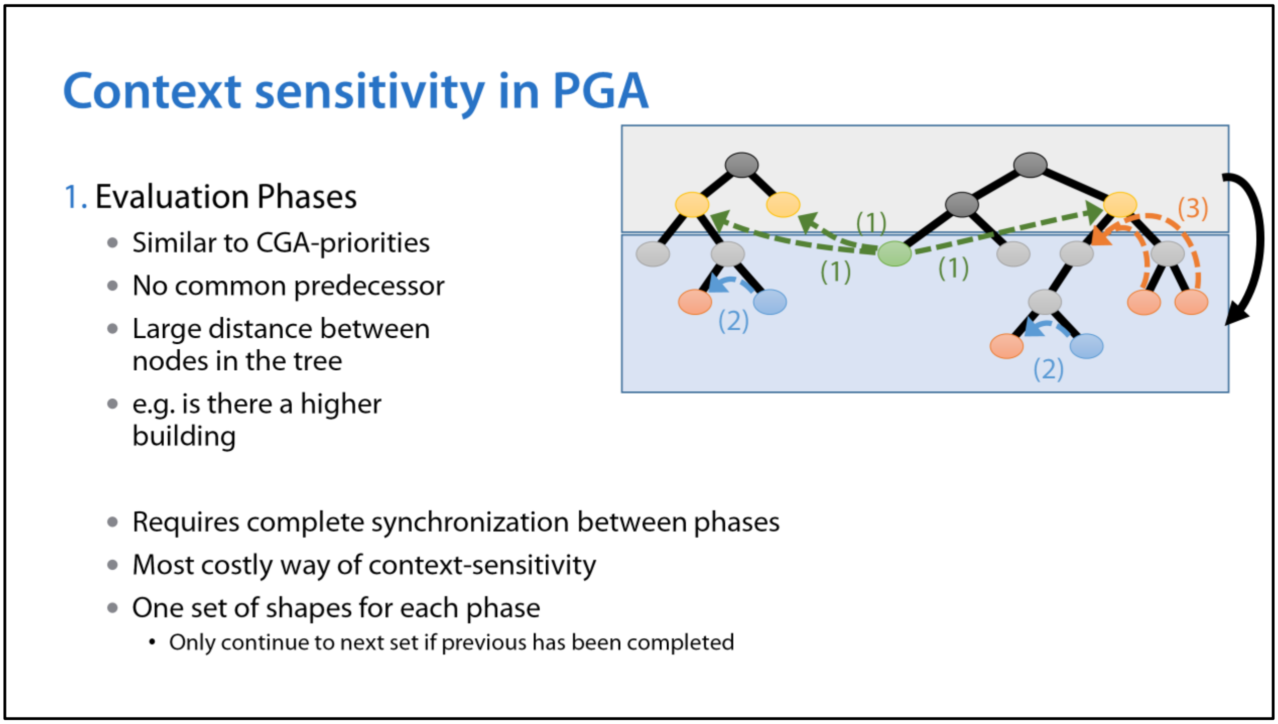
Generation of geometry on the GPU

- Steinberger et al. EG 2014 (PGA)
- Support for context-sensitive queries on the GPU
- Rule scheduling to reduce kernel overhead
- Rule grouping to reduce divergence & global memory accesses
- Operator level parallelism to increase performance
- Alternative Solution: Marvie et al. 2012



Context sensitivity in PGA

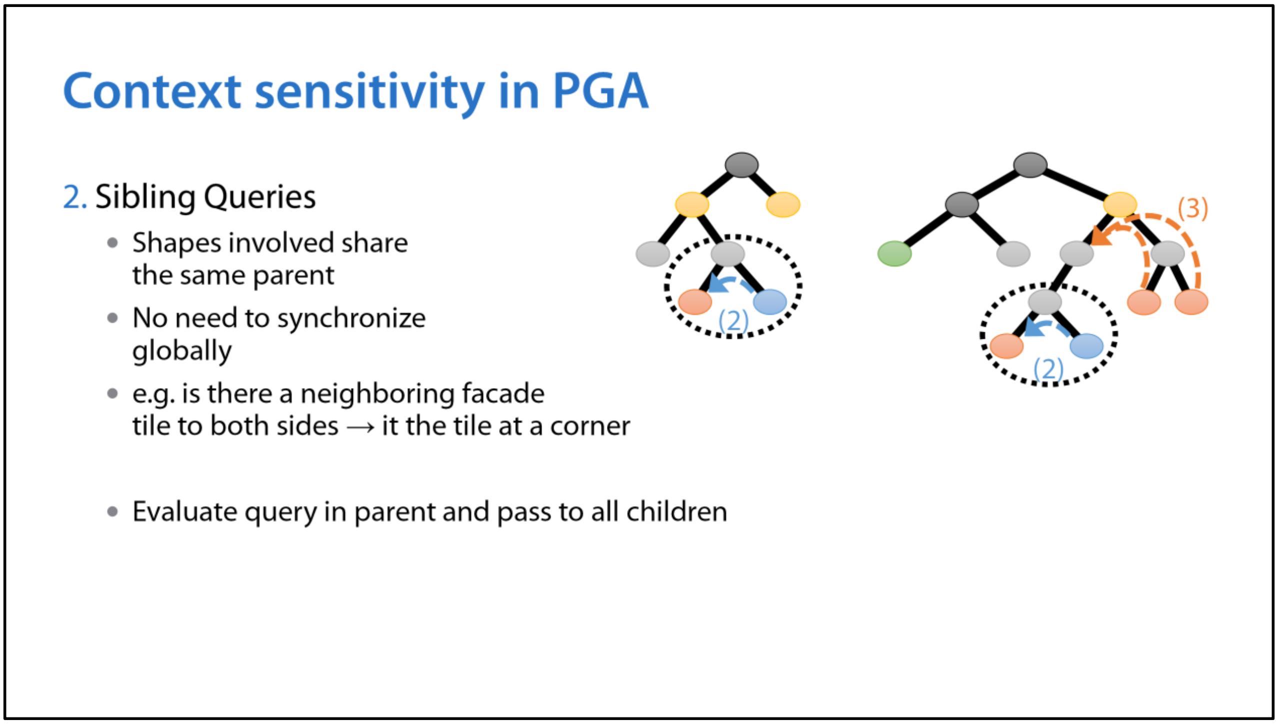
- Three mechanisms to express dependencies



Context sensitivity in PGA

1. Evaluation Phases
   - Similar to CGA-priorities
   - No common predecessor
   - Large distance between nodes in the tree
   - e.g. is there a higher building

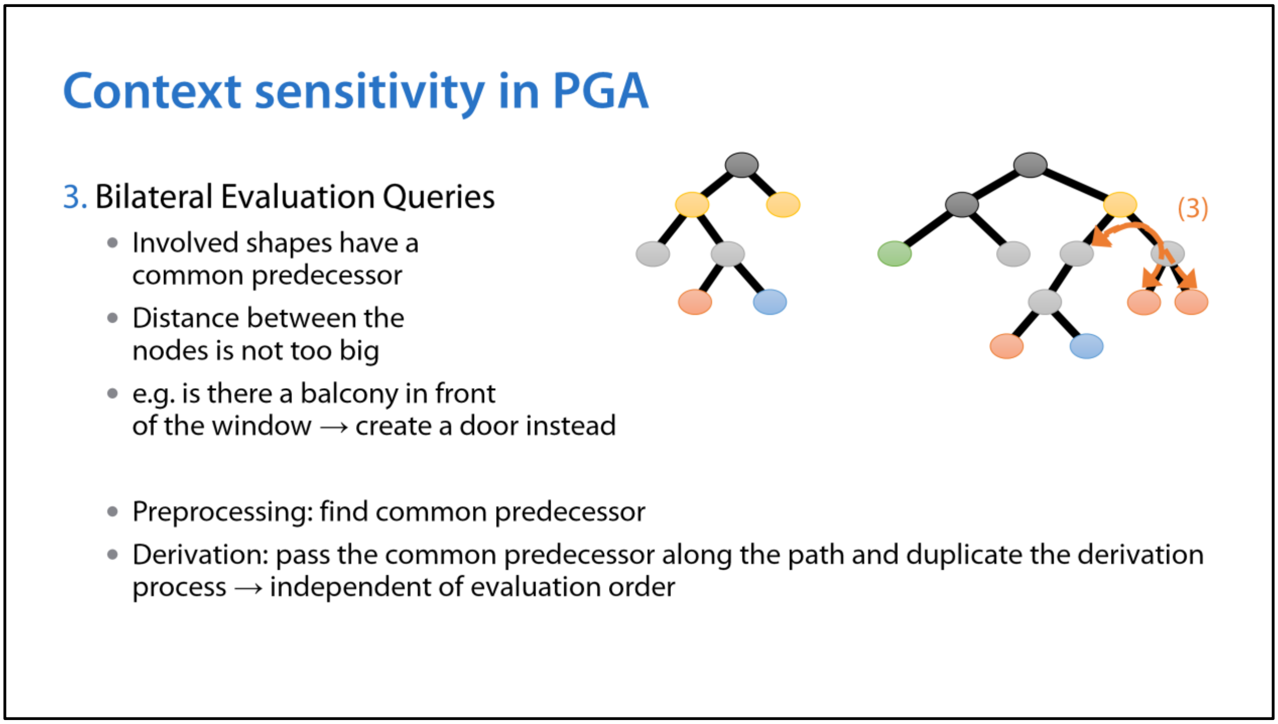
- Requires complete synchronization between phases
- Most costly way of context-sensitivity
- One set of shapes for each phase
  - Only continue to next set if previous has been completed



Context sensitivity in PGA

2. Sibling Queries
   - Shapes involved share the same parent
   - No need to synchronize globally
   - e.g. is there a neighboring facade tile to both sides → it the tile at a corner

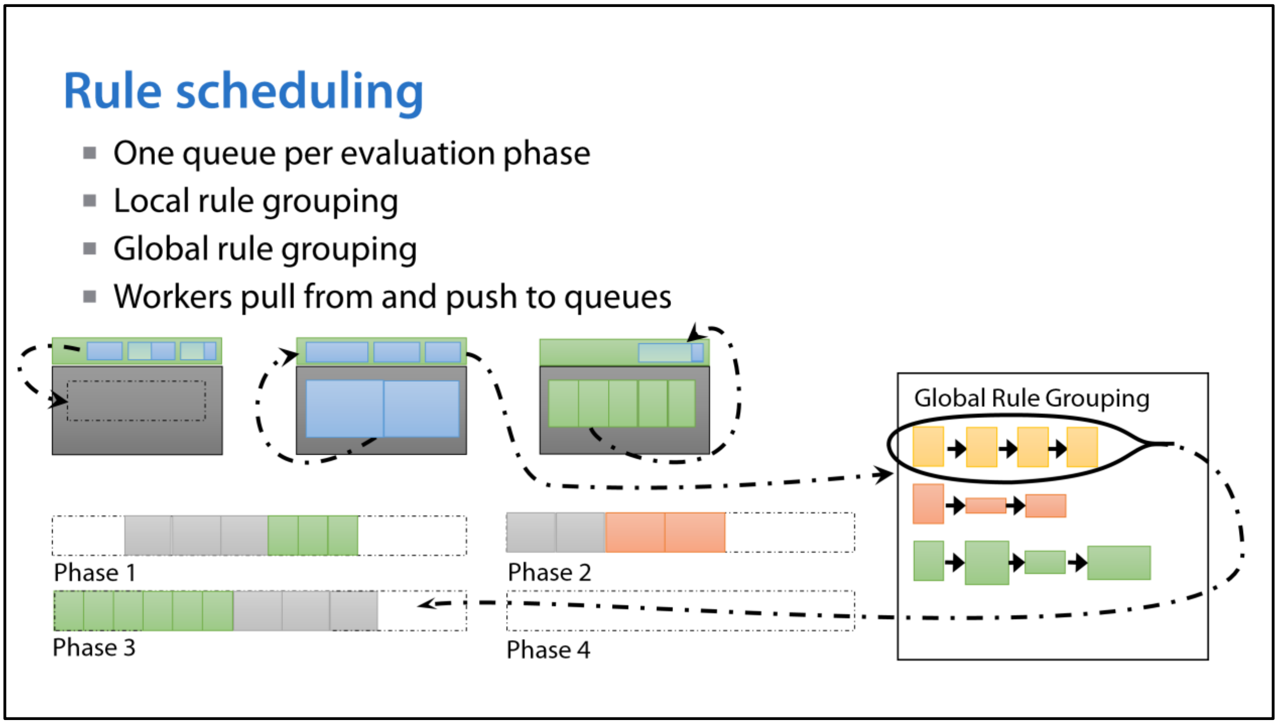
   - Evaluate query in parent and pass to all children



Context sensitivity in PGA

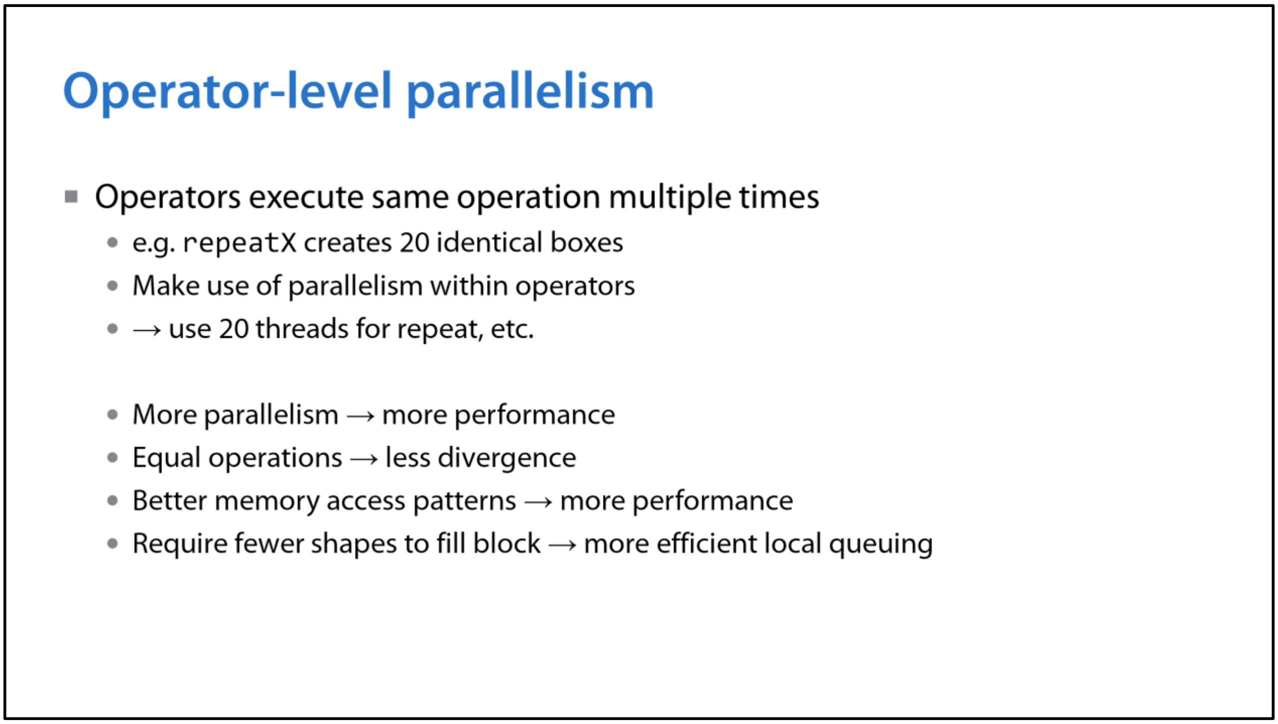
3. Bilateral Evaluation Queries
   - Involved shapes have a common predecessor
   - Distance between the nodes is not too big
   - e.g. is there a balcony in front of the window → create a door instead

- Preprocessing: find common predecessor
- Derivation: pass the common predecessor along the path and duplicate the derivation process → independent of evaluation order



Rule scheduling

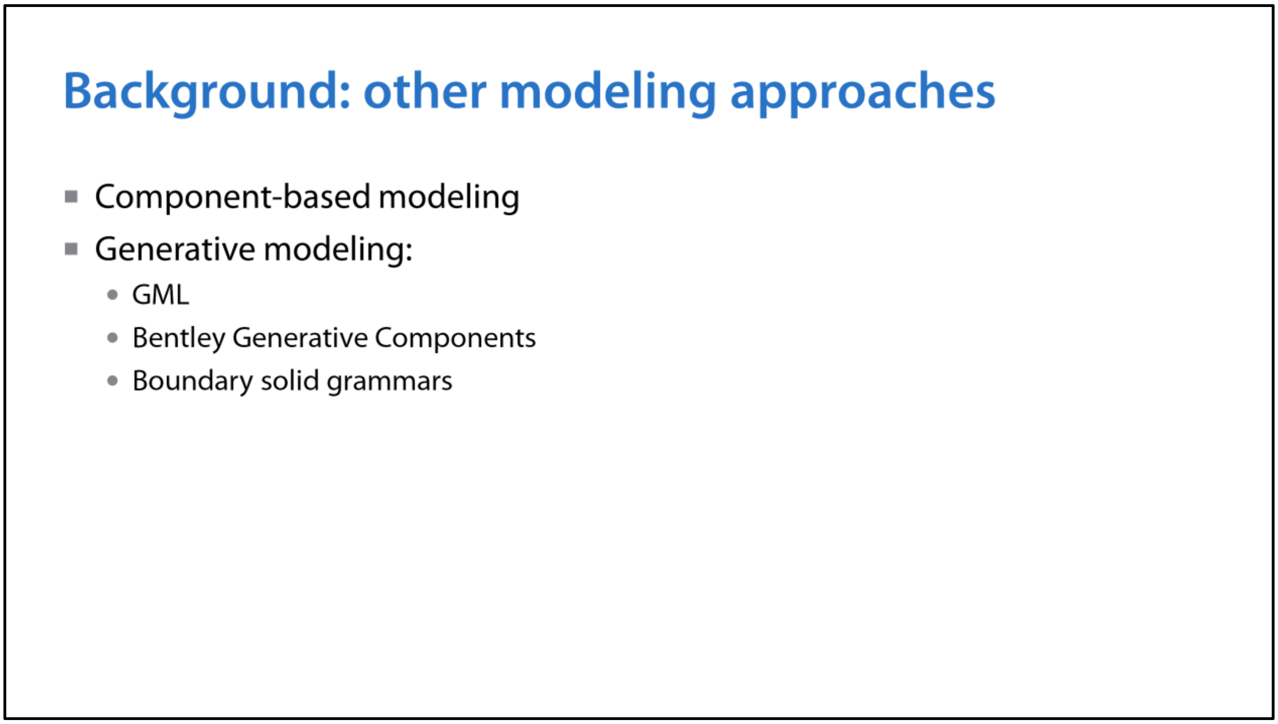
- One queue per evaluation phase
- Local rule grouping
- Global rule grouping
- Workers pull from and push to queues



Operator-level parallelism

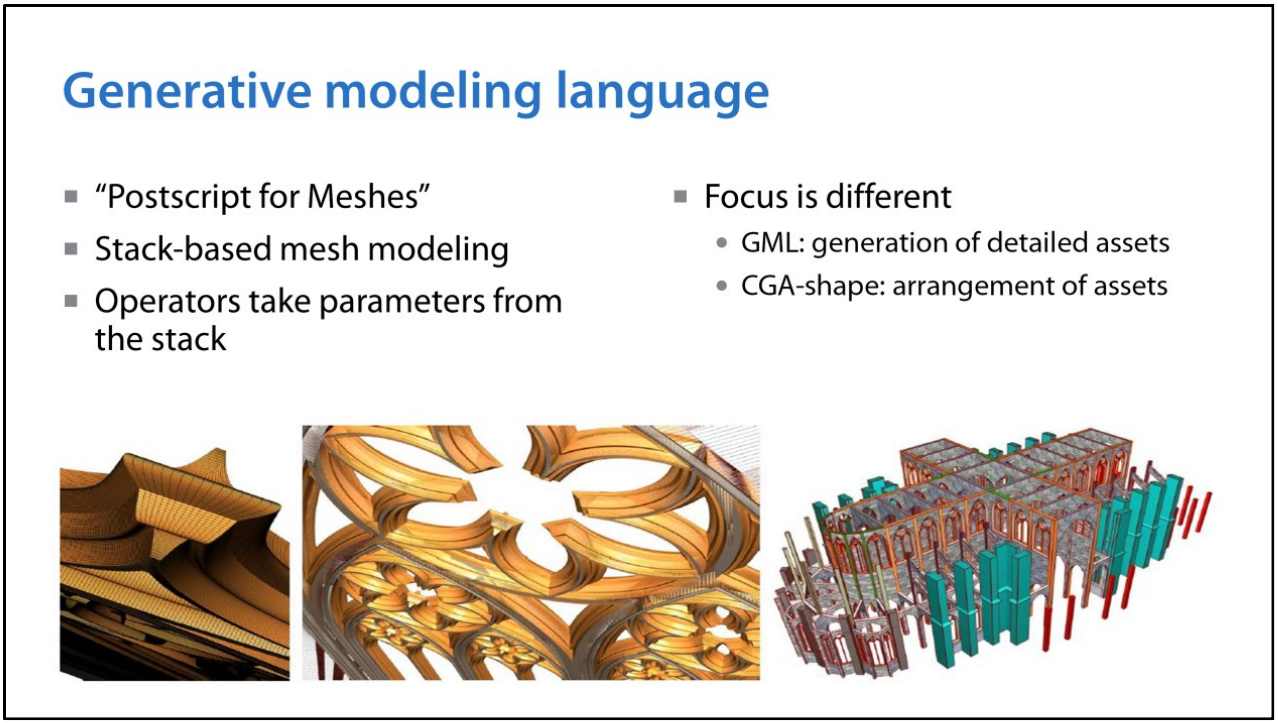
- Operators execute same operation multiple times
  - e.g. `repeatX` creates 20 identical boxes
  - Make use of parallelism within operators
  - → use 20 threads for repeat, etc.

  - More parallelism → more performance
  - Equal operations → less divergence
  - Better memory access patterns → more performance
  - Require fewer shapes to fill block → more efficient local queuing



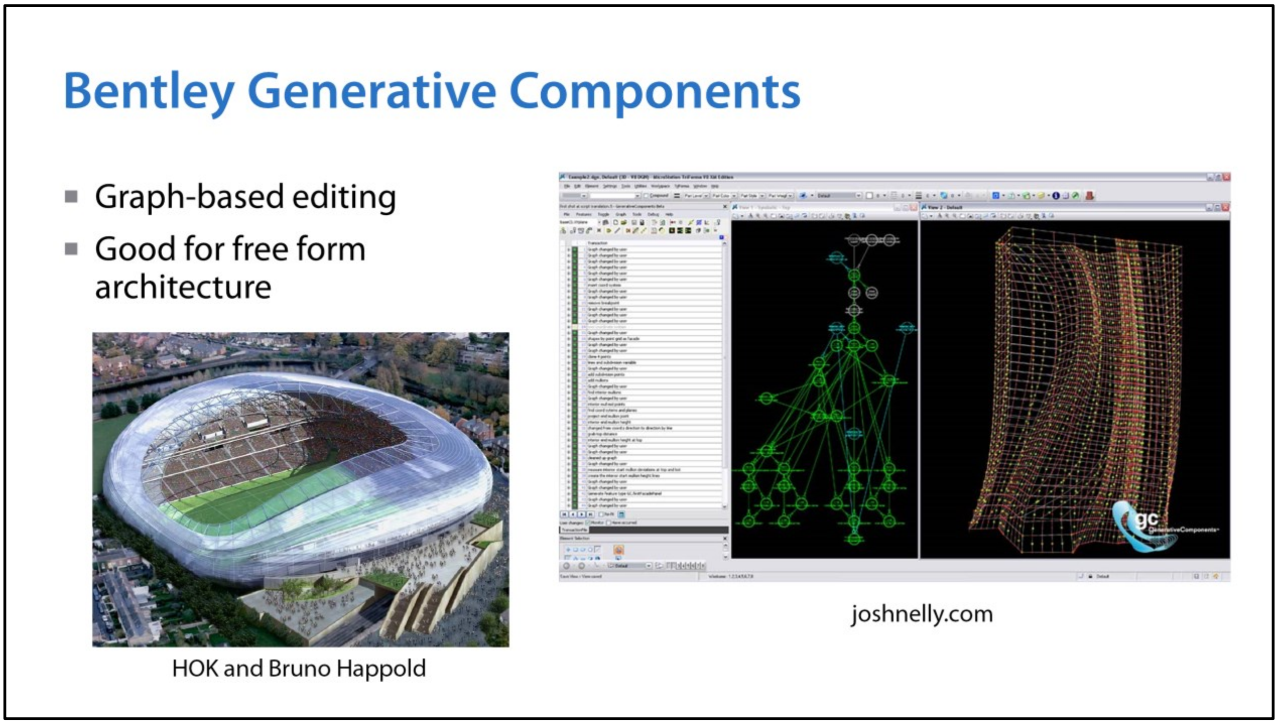
Background: other modeling approaches

- Component-based modeling
- Generative modeling:
  - GML
  - Bentley Generative Components
  - Boundary solid grammars



Generative modeling language

- “Postscript for Meshes”
- Stack-based mesh modeling
- Operators take parameters from the stack
- Focus is different
  - GML: generation of detailed assets
  - CGA-shape: arrangement of assets



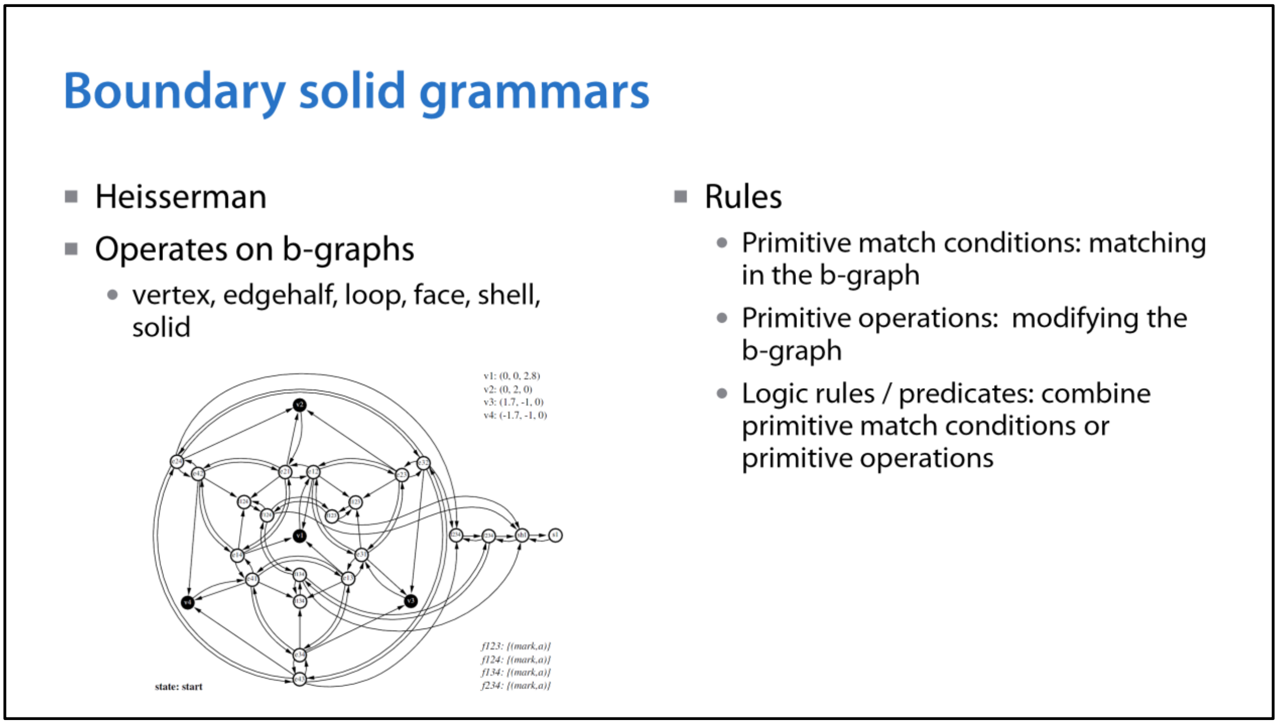
Bentley Generative Components

- Graph-based editing
- Good for free form architecture

HOK and Bruno Happold

joshnelly.com

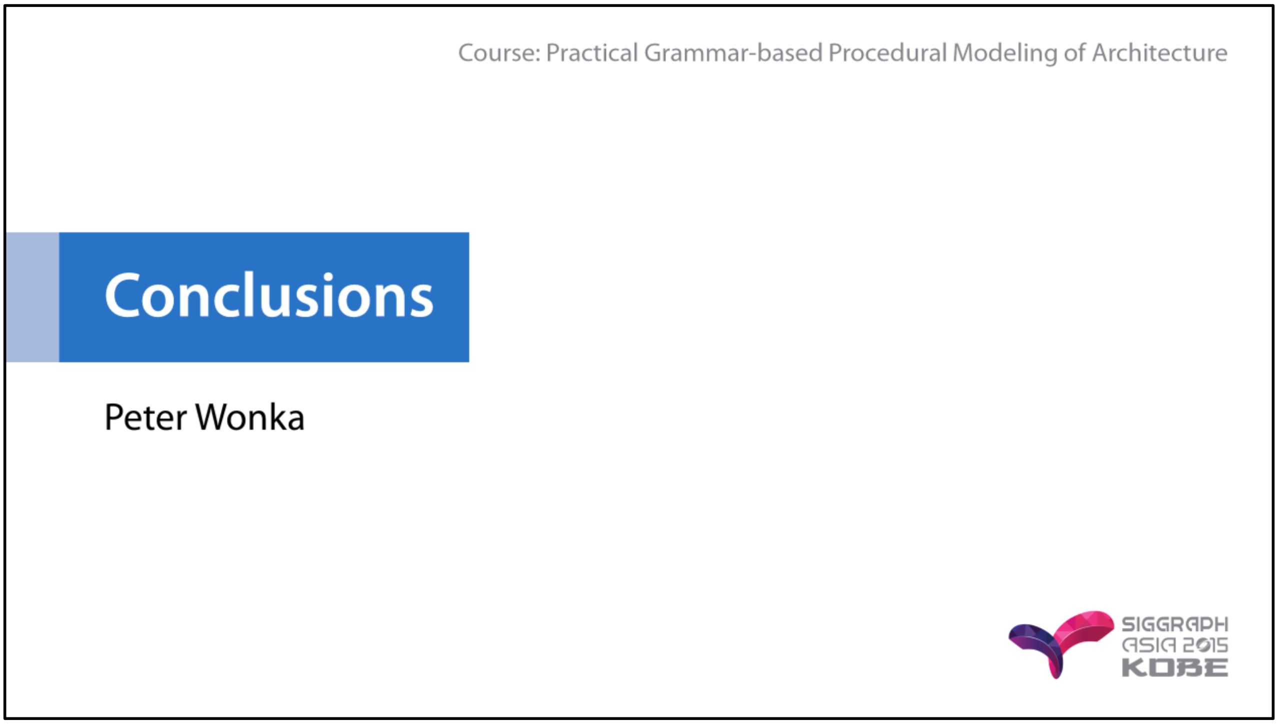
Bentley Generative Components is popular for modeling free-form architecture. For example, stadiums.



Boundary solid grammars

- Heisserman
- Operates on b-graphs
  - vertex, edgehalf, loop, face, shell, solid
- Rules
  - Primitive match conditions: matching in the b-graph
  - Primitive operations: modifying the b-graph
  - Logic rules / predicates: combine primitive match conditions or primitive operations

6 Conclusions

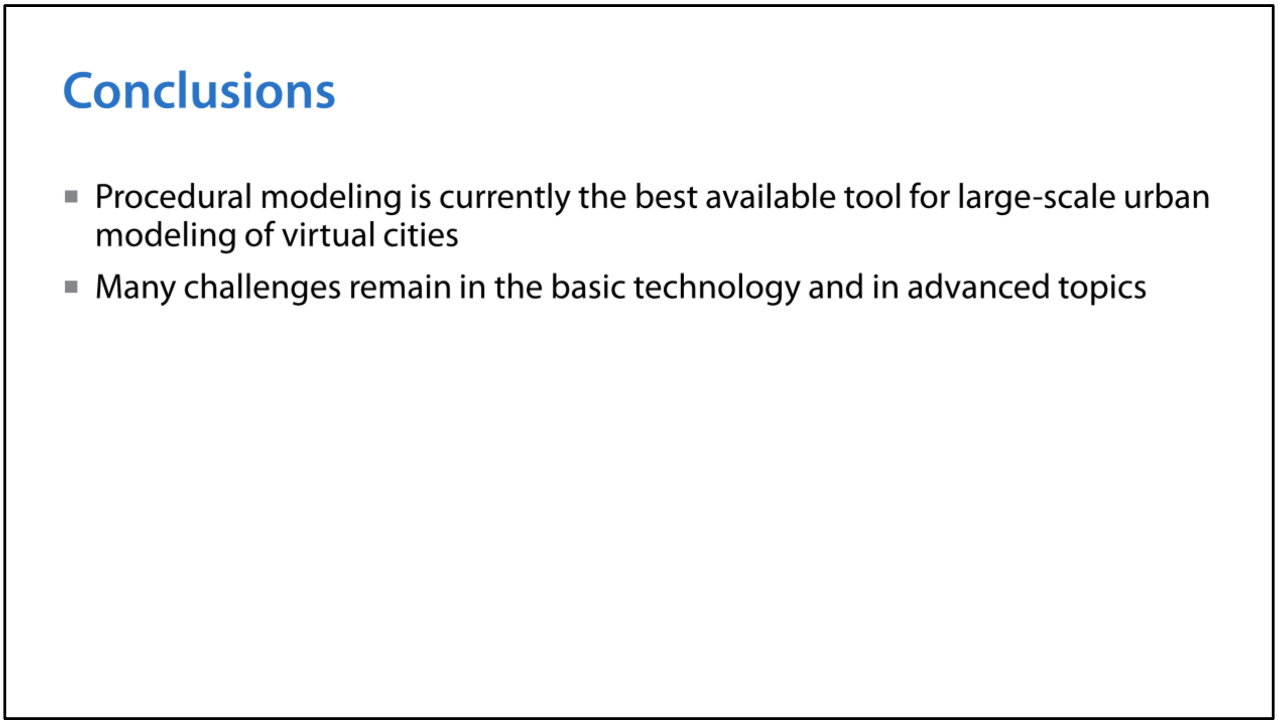


Course: Practical Grammar-based Procedural Modeling of Architecture

Conclusions

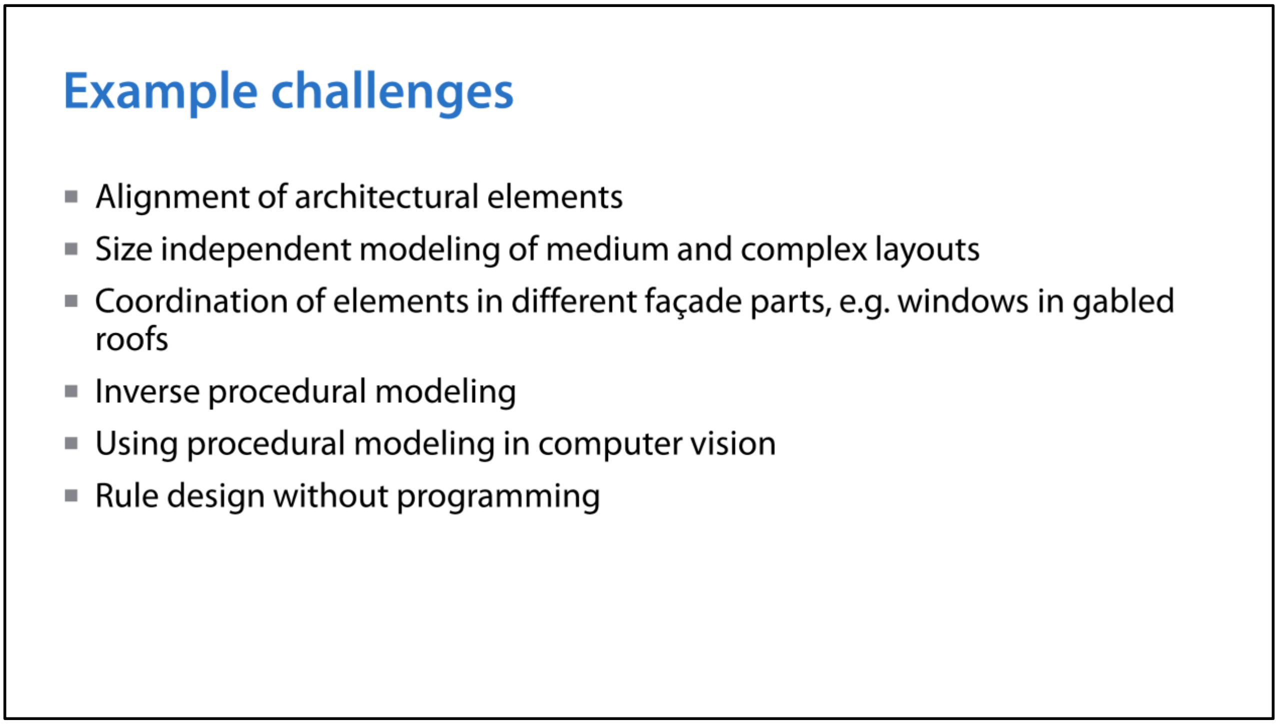
Peter Wonka

SIGGRAPH
ASIA 2015
KOBE



Conclusions

- Procedural modeling is currently the best available tool for large-scale urban modeling of virtual cities
- Many challenges remain in the basic technology and in advanced topics



Example challenges

- Alignment of architectural elements
- Size independent modeling of medium and complex layouts
- Coordination of elements in different façade parts, e.g. windows in gabled roofs
- Inverse procedural modeling
- Using procedural modeling in computer vision
- Rule design without programming

Bibliography

Aliaga, D. G., Rosen, P. A., and Bekins, D. R. 2007. Style grammars for interactive visualization of architecture. *IEEE Transactions on Visualization and Computer Graphics*, *13*, 4, 786–797.

Barroso, S., Besuievsky, G., and Patow, G. 2013. Visual copy & paste for procedurally modeled buildings by ruleset rewriting. *Computers & Graphics*, *37*, 4, 238–246.

Beneš, B., Šťava, O., Měch, R., and Miller, G. 2011. Guided procedural modeling. *Computer Graphics Forum*, *30*, 2, 325–334.

Besuievsky, G. and Patow, G. 2013. Customizable LoD for procedural architecture. *Computer Graphics Forum*, *32*, 8, 26–34.

Bokeloh, M., Wand, M., and Seidel, H.-P. 2010. A connection between partial symmetry and inverse procedural modeling. *ACM Transactions on Graphics*, *29*, 4, 104:1–104:10.

Buron, C., Marvie, J.-E., and Gautron, P. 2013. GPU roof grammars. In *Eurographics 2013: Short Papers*, pp. 57–60.

Dang, M., Lienhard, S., Ceylan, D., Neubert, B., Wonka, P., and Pauly, M. 2015. Interactive design of probability density functions for shape grammars. *ACM Transactions on Graphics*, *34*, 6, 206:1–206:13.

Haegler, S., Wonka, P., Müller Arisona, S., Gool, L. V., and Müller, P. 2010. Grammar-based encoding of facades. *Computer Graphics Forum*, *29*, 4, 1479–1487.

Havemann, S. 2005. *Generative Mesh Modeling*. Ph.D. thesis, TU Braunschweig.

Heisserman, J. A. 1991. *Generative Geometric Design and Boundary Solid Grammars*. Ph.D. thesis, Carnegie Mellon University.

Hohmann, B., Havemann, S., Krispel, U., and Fellner, D. 2010. A GML shape grammar for semantically enriched 3D building models. *Computers & Graphics*, *34*, 4, 322–334.

Kelly, T., Wonka, P., and Müller, P. 2015. Interactive dimensioning of parametric models. *Computer Graphics Forum*, *34*, 2, 117–129.

Krecklau, L., Born, J., and Kobbelt, L. 2013. View-dependent realtime rendering of procedural facades with high geometric detail. *Computer Graphics Forum*, *32*, 2, 479–488.

Krecklau, L. and Kobbelt, L. 2011a. Procedural modeling of interconnected structures. *Computer Graphics Forum*, *30*, 2, 335–344.

Krecklau, L. and Kobbelt, L. 2011b. Realtime compositing of procedural facade textures on the GPU. In *Proceedings of 3D-ARCH 2011*, pp. 177–184.

Krecklau, L. and Kobbelt, L. 2012. Interactive modeling by procedural high-level primitives. *Computers & Graphics*, *36*, 5, 376–386.

Krecklau, L., Pavic, D., and Kobbelt, L. 2010. Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum*, *29*, 8, 2291–2303.

Kuang, Z., Chan, B., Yu, Y., and Wang, W. 2013. A compact random-access representation for urban modeling and rendering. *ACM Transactions on Graphics*, *32*, 6, 172:1–172:11.

Larive, M. and Gaildrat, V. 2006. Wall grammar for building generation. In *Proceedings of GRAPHITE 2006*, pp. 429–437.

Leblanc, L., Houle, J., and Poulin, P. 2011. Component-based modeling of complete buildings. In *Proceedings of Graphics Interface 2011*, pp. 87–94.

Lienhard, S., Specht, M., Neubert, B., Pauly, M., and Müller, P. 2014. Thumbnail galleries for procedural models. *Computer Graphics Forum*, *33*, 2, 361–370.

Liew, H. 2004. *SGML: A Meta-Language for Shape Grammars*. Ph.D. thesis, Massachusetts Institute of Technology.

Lipp, M., Wonka, P., and Wimmer, M. 2008. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics*, *27*, 3, 102:1–102:10.

Marks, J., Andalman, B., Beardsley, P. A., Freeman, W., Gibson, S., Hodgins, J., Kang, T., Mirtich, B., Pfister, H., Ruml, W., Ryall, K., Seims, J., and Shieber, S. 1997. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of SIGGRAPH 97*, pp. 389–400.

Marvie, J.-E., Buron, C., Gautron, P., Hirtzlin, P., and Sourimant, G. 2012. GPU shape grammars. *Computer Graphics Forum*, *31*, 7, 2087–2095.

Marvie, J.-E., Gautron, P., Hirtzlin, P., and Sourimant, G. 2011. Render-time procedural per-pixel geometry generation. In *Proceedings of Graphics Interface 2011*, pp. 167–174.

Měch, R. and Prusinkiewicz, P. 1996. Visual models of plants interacting with their environment. In *Proceedings of SIGGRAPH 96*, pp. 397–410.

Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. V. 2006. Procedural modeling of buildings. *ACM Transactions on Graphics*, *25*, 3, 614–623.

Müller, P., Zeng, G., Wonka, P., and Gool, L. V. 2007. Image-based procedural modeling of facades. *ACM Transactions on Graphics*, *26*, 3, 85:1–85:9.

Musialski, P., Wonka, P., Aliaga, D. G., Wimmer, M., van Gool, L., and Purgathofer, W. 2013. A survey of urban reconstruction. *Computer Graphics Forum*, *32*, 6, 146–177.

Parish, Y. I. H. and Müller, P. 2001. Procedural modeling of cities. In *Proceedings of SIGGRAPH 2001*, pp. 301–308.

Patow, G. 2012. User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications*, *32*, 2, 66–75.

Prusinkiewicz, P., James, M., and Měch, R. 1994. Synthetic topiary. In *Proceedings of SIGGRAPH 94*, pp. 351–358.

Prusinkiewicz, P. and Lindenmayer, A. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York.

Prusinkiewicz, P., Mündermann, L., Karwowski, R., and Lane, B. 2001. The use of positional information in the modeling of plants. In *Proceedings of SIGGRAPH 2001*, pp. 289–300.

Schwarz, M. and Müller, P. 2015. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, *34*, 4, 107:1–107:12.

Schwarz, M. and Wonka, P. 2014. Procedural design of exterior lighting for buildings with complex constraints. *ACM Transactions on Graphics*, *33*, 5, 166:1–166:16.

Silva, P. B., Eisemann, E., Bidarra, R., and Coelho, A. 2015. Procedural content graphs for urban modeling. *International Journal of Computer Games Technology*, *2015*, 808904:1–808904:15.

Silva, P. B., Müller, P., Bidarra, R., and Coelho, A. 2013. Node-based shape grammar representation and editing. In *Proceedings of Fourth Workshop on Procedural Content Generation in Games*.

Smelik, R. M., Tutenel, T., Bidarra, R., and Benes, B. 2014. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, *33*, 6, 31–50.

Snyder, J. M. 1992. *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design using Interval Analysis*. Academic Press, San Diego.

Šťava, O., Beneš, B., Měch, R., Aliaga, D. G., and Krištof, P. 2010. Inverse procedural modeling by automatic generation of L-systems. *Computer Graphics Forum*, *29*, 2, 665–674.

Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Wonka, P., and Schmalstieg, D. 2014a. Parallel generation of architecture on the GPU. *Computer Graphics Forum*, *33*, 2, 73–82.

STEINBERGER, M., KENZEL, M., KAINZ, B., WONKA, P., AND SCHMALSTIEG, D. 2014b. On-the-fly generation and rendering of infinite cities on the GPU. *Computer Graphics Forum*, *33*, 2, 105–114.

STINY, G. 1980. Introduction to shape and shape grammars. *Environment and Planning B*, *7*, 3, 343–351.

STINY, G. 1982. Spatial relations and grammars. *Environment and Planning B*, *9*, 1, 113–114.

STINY, G. 2006. *Shape: Talking about Seeing and Doing.* MIT Press.

STINY, G. AND GIPS, J. 1972. Shape grammars and the generative specification of painting and sculpture. In *Information Processing 71*, pp. 1460–1465.

TALTON, J. O., LOU, Y., LESSER, S., DUKE, J., MĚCH, R., AND KOLTUN, V. 2011. Metropolis procedural modeling. *ACM Transactions on Graphics*, *30*, 2, 11:1–11:14.

THALLER, W., KRISPEL, U., HAVEMANN, S., AND FELLNER, D. W. 2012. Implicit nested repetition in dataflow for procedural modeling. In *Proceedings of Computation Tools 2012*, pp. 45–50.

THALLER, W., KRISPEL, U., ZMUGG, R., HAVEMANN, S., AND FELLNER, D. W. 2013. Shape grammars on convex polyhedra. *Computers & Graphics*, *37*, 6, 707–717.

VANEGAS, C. A., ALIAGA, D. G., WONKA, P., MÜLLER, P., WADDELL, P., AND WATSON, B. 2010. Modelling the appearance and behaviour of urban spaces. *Computer Graphics Forum*, *29*, 1, 25–42.

VANEGAS, C. A., GARCIA-DORADO, I., ALIAGA, D. G., BENES, B., AND WADDELL, P. 2012. Inverse design of urban procedural models. *ACM Transactions on Graphics*, *31*, 6, 168:1–168:11.

WATSON, B., MÜLLER, P., WONKA, P., SEXTON, C., VERYOVKA, O., AND FULLER, A. 2008. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications*, *28*, 3, 18–26.

WHITING, E., OCHSENDORF, J., AND DURAND, F. 2009. Procedural modeling of structurally-sound masonry buildings. *ACM Transactions on Graphics*, *28*, 5, 112:1–112:9.

WONKA, P., WIMMER, M., SILLION, F. X., AND RIBARSKY, W. 2003. Instant architecture. *ACM Transactions on Graphics*, *22*, 3, 669–677.

WU, F., YAN, D.-M., DONG, W., ZHANG, X., AND WONKA, P. 2014. Inverse procedural modeling of facade layouts. *ACM Transactions on Graphics*, *33*, 4, 121:1–121:10.