Chapter 35
CityEngine: An Introduction to Rule-Based Modeling

Tom Kelly

Abstract CityEngine is a rule-based urban modeling software package. It offers a flexible pipeline to transform 2D data into 3D urban models. Typical applications include processing 2D urban cartographic geographic information system (GIS) data to create a detailed 3D city model, creating a detailed visualization of a proposed development, or exploring the design space of a potential project. The rule-based core of Esri's CityEngine has some unique advantages: Huge cities can be created as easily as small ones, while the quality of the models is consistent throughout. Additionally, this rule-based approach means that large design spaces can be explored quickly, interactively, and analytically compared. Such advantages must be carefully balanced against the increased time to create and parameterize the rules and the sometimes stylistic or approximate models created; coming from more traditional workflows, CityEngine's pipeline can be initially overwhelming. We introduce the principal workflows and the flexibility they afford, sketch the procedural programming language used, and discuss the export pathways available.

35.1 3D: One Better than 2D

3D technologies are revolutionizing the way we plan, understand, communicate, and document our urban environments. Revolutions are, however, rarely easy; there are numerous issues and challenges around this transition from 2D to 3D toolchains.

Reading 2D plans and maps is often challenging because they are one dimension short of the 3D world we live in. The 3D data must be encoded using various tricks and conventions, such as contour lines, elevation diagrams, symbols, and shading. This is because there is more information in the 3D world than 2D plans contain. Technology now enables us to efficiently record, model, and plot in 3D. Collecting and sharing this 3D information has been, until recently, difficult and prohibitively

T. Kelly (✉)
University of Leeds, Leeds, UK
e-mail: twakelly@gmail.com

W. Shi et al. (eds.), *Urban Informatics*, The Urban Book Series,
https://doi.org/10.1007/978-981-15-8983-6_35

expensive. As various technologies such as commodity 3D CAD and photogrammetric reconstruction have matured, we are able to accurately construct virtual 3D models of our 3D world.

At the same time as making our data more accurate, 3D models make our data more accessible. While it has always been possible to create physical scale models of our environments, these are expensive, difficult to transport or share, and bulky to store. Technologies such as immersive virtual and augmented realities (VR, AR, often summarized as XR) allow anyone from children to city planners to understand complex designs by exploring them at real-world scales. 3D tools such as physical simulation (solar potential, window modeling) and viewpoint rendering help engineers design empirically better environments; because we are able to explore our design spaces more quickly, we understand them faster, produce better designs, and better comprehend any issues.

However, 3D modeling is difficult. The de facto 3D representation is the mesh. This is a set of corners (vertices) placed in 3D space, between which we create triangles. By creating many thousands of such triangles, we can build representations of complex 3D environments. We may even choose to apply colors or texture to each triangle.

There are many tools available for creating these polygonal meshes. Traditional manual 3D modeling tools offer a way to create multiple triangles at a time by creating more complex primitives (spheres, cubes, curves, surfaces, extrusions, etc.). Such manual tools include Autodesk Maya (2019), Trimble SketchUp (2019), or Blender (2019). Even though these manual tools have become incredibly sophisticated and general, they still require users to spend a lot of time positioning and editing triangles and primitives. For our use cases, we might imagine our long-suffering artist being employed to position a spherical doorknob on every rectangular front door, of every building, in the urban area we are modeling.

What we would rather do is to create a rule which encodes "attach a sphere to every front door". Luckily, computers are rather good at these repetitive tasks—if we can find a way to explain to them what to do. In this chapter, we introduce one way to instruct them: rule-based modeling. In particular, we will dive deeply into a particular modeling system: Esri's CityEngine. Such modeling systems offer tools to procedurally generate 3D meshes from systems of rules—they are able to create models with millions of vertices in seconds.

It is here that we see another advantage of working with virtual, rather than physical, 3D models. Computer programs can follow rules to create and manipulate virtual polygonal mesh models superhumanly quickly and accurately. We can repeatedly change the rules and view and explore the resulting environments on screen, in virtual reality, or physically produce them using a 3D printer. To perform the same changes in a physical 3D model would take many lifetimes.

35.2 2D Shapes + Rules = 3D Models

Because of the hierarchical, systematic, and often repetitive nature of urban environments, rule-based city modeling has been a driving force for general procedural modeling in general. We note in passing that other rule-based systems have been wildly successful in other domains. Of note are commercial systems such as SpeedTree (2019) for the rapid generation of trees and forests and Grome (Wikipedia 2019) for creating terrains and landscapes. For each different domain, different techniques and rules are appropriate. In CityEngine, as we will see, the rules and the operations they use have been carefully curated to allow rapid and accurate modeling of buildings and streets.

Before deciding to use a rule-based modeling pipeline, it is important to weigh the advantages and disadvantages against more traditional manual modeling pipelines. For smaller or more complex models, manual modeling may be faster and cheaper; the time to create the rules may be larger than the time that would be taken to perform the manual modeling. Rule-based modeling is particularly difficult for complex geometries where many decisions are involved in placement and evaluation. Translating each decision into a rule and ensuring that the decisions interact appropriately in all circumstances can be time-consuming. We note that many of the explanatory examples in this chapter would be more quickly created using manual modeling tools—only when scaling up to larger areas does rule-based modeling reward the time invested in creating the rules.

Writing rule files is a new skill that must be taught, studied, and maintained like any other. Because it is a newer technology, finding qualified personnel can be more difficult, especially because they may need a background in urban design, a basic knowledge of linear algebra, as well as the ability to (en)code our rules in a programming language.

These caveats aside, rule-based modeling is able to offer a flexible, quick, and responsive toolchain for quickly developing urban scenarios ranging from single building modeling, campus-scale designs, up to neighborhood and city-scale simulation. Once the rules are available, a large quantity of geometry can be created easily and quickly. Changes and modifications to scenarios can be made in real time. Both the level of detail ("do we draw chimneys on the buildings?", "do we draw roofs?"), the presentation format (Webviewer, VR), and the rule attributes ("how high is this building?") can be updated over an entire city at once, all thanks to rule-based modeling.

Esri's CityEngine is a software system for rule-based modeling in the urban domain. It provides a visual environment to apply rules, create new rules, and inspect the results. The historical context of CityEngine was that it was acquired by Esri during their transition from a 2D cartography company to a provider of 3D solutions. As witnessed by ArcGIS Pro, this transition has created a massively powerful pipeline with support for all the major industry formats. This business context underpins the CityEngine workflow—2D shapes are imported into the system, where rules are used to convert them to 3D models. These models are the 3D output which we

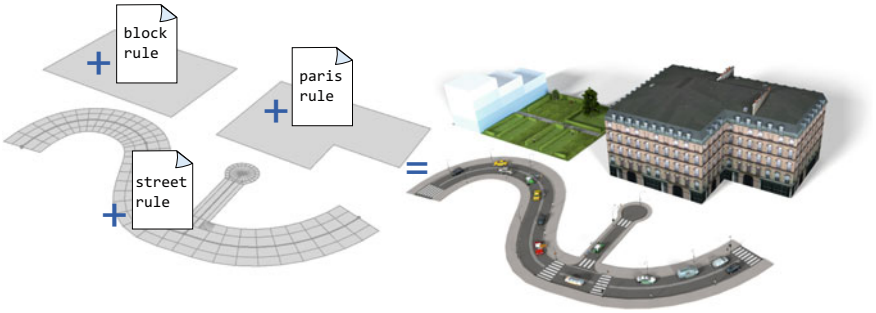


Fig. 35.1 The central paradigm of CityEngine is to apply rules to shapes (gray, left) to create 3D models (right). This approach is able to create a large variety of rule-driven models

may view in CityEngine or export to the Web or VR. Thus, the central process for modeling in CityEngine is to apply rules to shapes to create models (Fig. 35.1).

A CGA rule is a text file containing a list of instructions. In Fig. 35.2, we introduce a simple rule which extrudes a shape into a model of a 3D prism. While this rule only contains five lines of code, complex rule files can be thousands of lines long.

This chapter aims to be a broad introductory tour of the system with a deep dive into various implementation topics. We continue to describe shapes, the rules, analysis tools, and export paths from CityEngine. After reading this chapter, the kinesthetic learner is encouraged to spend a few days working through the CityEngine tutorials provided by Esri (2019a). Similarly, Esri's online documentation is an invaluable source of technical details (Esri 2019b).

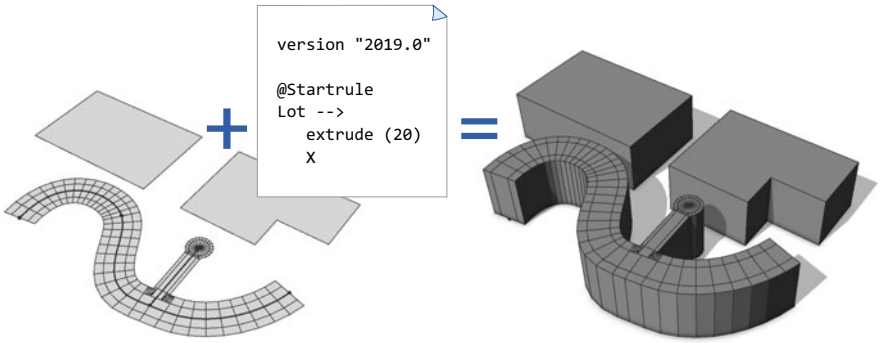


Fig. 35.2 A simple CGA rule file (center) is applied to several different shapes (left) to create the associated 3D models (right). This rule creates a prism of height 20 m over the shape

35.3 On the (Many) Origins of Shapes

CityEngine provides two workflows to instantly create entire cities with very little user input. The City Wizard (File → New… → CityEngine → CityWizard) uses an entirely procedural workflow to create an impressive quantity of shapes with complex rules in a few clicks. Of course, the resulting city is entirely fictional; if we wish instead to use an entirely data-driven set of shapes, we may use the Map Import (File → get Map Data…). This tool downloads satellite images, height maps, lot footprints, and street networks, to create shapes and terrain for a real-world area (Fig. 35.3). However, because there is no common data source for building rules, only simple rules are provided. Both the City Wizard and Map Import use shapes to model entire cities quickly but leave us with limited control over the shapes and rules. We continue to examine more controlled ways to create shapes.

Shapes are usually 2D polygons lying on the ground. Much of CityEngine's utility and complexity is driven by the different ways to create shapes. The various sources for shapes provide an overview of the different modeling workflows available in CityEngine:

- To create a 3D model of an existing area, we may use a collection of building lots from a geospatial data source (including FileGDB, DXF, Shapefile, or OBJ) as shapes.
- To plan a new urban area, we may draw our own shapes, for example by adding each corner of each lot at a time. The simplest way to create a shape is to use the Rectangular Shape Creation tool, which allows clicking and dragging to position two corners of a rectangle on the floor plane. To increase the accuracy, we may trace the outline of these shapes from images imported into CityEngine.

Fig. 35.3 A city created in 30 s using the Map Import functionality

- If we wish to use rules to add windows to the blank facades of a building, we could draw the building using the manual 3D modeling tools provided by CityEngine. This is an uncommon workflow because the shapes may not be horizontal. Such a workflow allows us to manually model a building and then apply rules only to specific façades. CityEngine has a range of tools for manual shape modeling, including rectangular, polygonal, and circle generation. Markus Lipp created this modeling system to use intelligent extrusions to quickly and manually model urban forms (Lipp et al. 2014).
- When modeling a street network, we may import a street graph (formats supported include DXF, FileGDB, and OpenStreetMap) and use CityEngine's dynamic shape system to automatically create street shapes, blocks, and lot shapes between the streets. We continue to explore the dynamic shape system in greater depth.

35.3.1 Dynamic Shapes: Streets, Blocks, and Lots

Dynamic shapes use algorithms to approximate the forms that we see in our urban environments. Because of this, they are only simulated designs that match general characteristics (the range of building lot widths) but not specific measurements (the width of a particular lot). We describe them as dynamic because they are generated dynamically from the street graph; if you move a street intersection, the adjoining roads and blocks are automatically recalculated. The flexibility of CityEngine allows for combinations of these shape generation approaches—manual, data-driven, and dynamic—to be used together. For example, streets can be imported from a GIS data source and the blocks between the streets can be dynamically subdivided to lots, or an area of the city where GIS data exist for streets and lots can be augmented by adjacent dynamically generated streets and lots.

A street graph describes the streets in a street network. Over this graph, dynamic street shapes are created for sidewalks, junctions, and the street themselves, as shown in Fig. 35.4. The graph edges describe the center lines, and the nodes (where the edges meet) describe the street junctions.

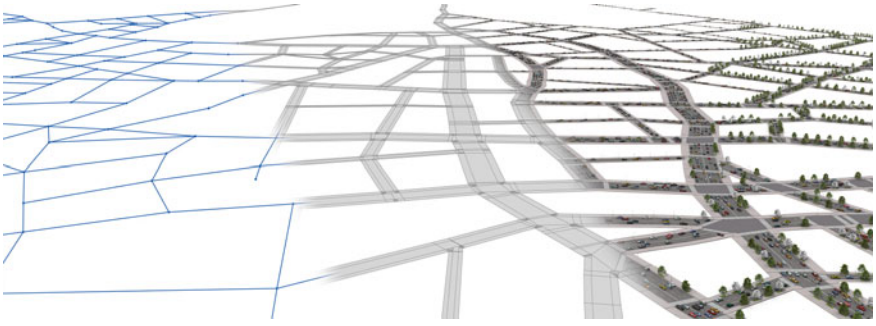


Fig. 35.4 Left: a blue street centerline graph; middle: the generated street shapes; right: 3D models generated by applying rules to the shapes

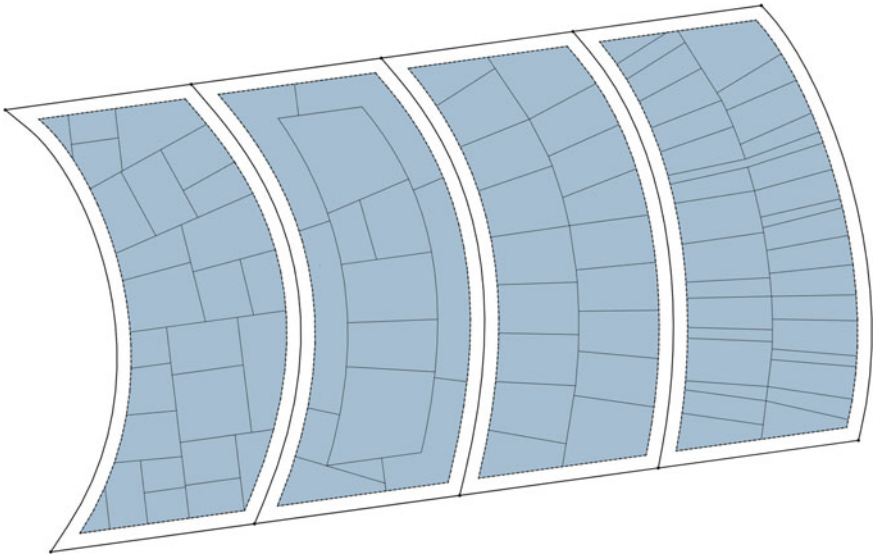


Fig. 35.5 Block subdivision algorithms used to create building lots. From left to right: *recursive*, *offset*, and *skeleton*. Far right: skeleton modified for a high irregularity and narrower lot width

Between streets, CityEngine dynamically generates blocks and from the blocks, lots. Generally, every loop of streets generates a block in its interior. The block contains a further selection of attributes which define its subdivision into lot shapes. The lot shape represents a parcel of land on which we will use rules to generate individual building models. When a block (or a street) is selected in CityEngine, the Inspector shows details about the object which drive the generation of the dynamic shapes. Block to lot subdivision algorithms are discussed by Vanegas et al. (2012) and are subdivided into two major categories: recursive subdivision and offsets. Each of these can be further controlled with attributes controlling on lot area, width, and variation, as in Fig. 35.5.

The generation sequence is an important part of the modeling paradigm used by CityEngine for dynamic shapes: Streets are created, between which blocks are found, and finally inside each block, lots are created. It is important to note this order when creating cityscapes and start with street creation before moving on to block and lot generation. This is because small changes in the street network will affect many blocks, whereas changing a block's subdivision settings will affect only the lots in the block. Similarly, changing a lot's rule or attributes will only affect the single lot's (building) model.

Remembering that our shapes will be the starting point for rules, it is also important to note the default starting rule names for each dynamic shape type. This name is used to automatically assign a start (initial) rule to the shape. For example, dragging a rule file onto a street's sidewalk shape will attempt to use the rule named Sidewalk (and taking no parameters), while the same file dragged onto a lot shape will use the rule Lot.

35.3.2 Graphs and Cities

The astute reader will notice that the street graphs (the street centerlines themselves) are not dynamic. The street graph contains the information required to dynamically create the other dynamic shapes. As we have come to expect, CityEngine provides manual, data-driven, and procedural approaches to creating street graphs.

Creating a street graph manually can be accomplished with the polygonal or freehand street creation tools. These allow graph vertices and edges to be created by clicking at corners or by sketching streets. The Edit Street tool can then be used to reposition vertices, curve streets, and adjust street or sidewalk widths.

An alternative to drawing street graphs directly is to import an existing graph from a GIS source. Supported formats include DXF, FileGDB, and OpenStreetMap. CityEngine can parse and map attributes such as street widths in some of these formats, which can avoid manual assignment with the Edit Street tool. Working with various data sources can take some experience because each has different properties such as distance between nodes or the presence of curved graph segments. To assist with working with these graphs, various tools are available to simplify a graph (*Graph → Simplify Graph…*), align the graph to the terrain (*Graph → Align Graph to Terrain*), or resolve crossing graph edges into bridges and underpasses (*Graph → Generate Bridges…*)

To create large street networks where there is no available GIS source, CityEngine provides the Grow Streets tool which creates a procedurally generated set of streets, as well as blocks and lots as described above. The origins of the street growth algorithms used are described in the paper by Parish and Müller (2001), although these have now advanced beyond the published details somewhat. In summary, self-sensitive L-Systems (Prusinkiewicz and Lindenmayer 2012) are employed to grow major and minor streets. Newly grown edges are snapped to attach to parts of the existing networks. By combining different patterns of growth for both the major and minor streets, a wide variety of different networks can be grown, illustrated in Fig. 35.6. The Grow Streets tool also allows the type of dynamic block subdivision to be specified.

Once a real street graph has been imported or synthetic graph has been grown, the Edit Street and Street Creation tools can be used to amend or fine-tune the data.

There are several use cases for graphs beyond their typical use of creating street models. Appropriate rules can be used to create various graph-like structures including walls, railroads, and power-lines as in Fig. 35.7.

We have seen an overview of the multitude of ways that CityEngine can be used to create different shapes; we continue to examine how we can obtain rules to transform our shapes into 3D models.

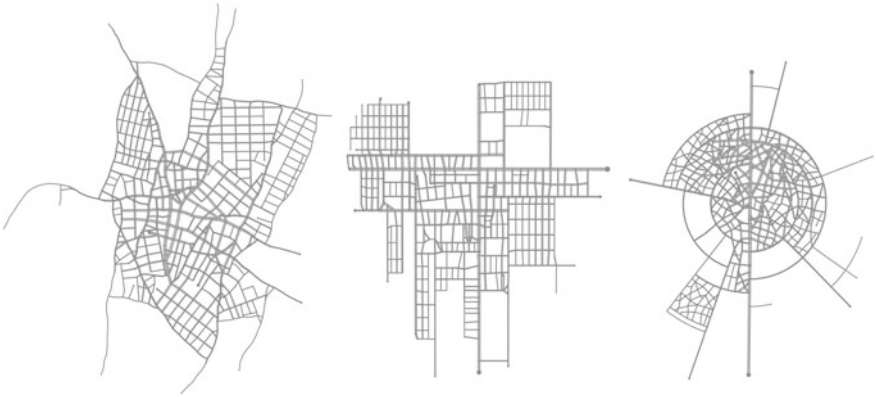


Fig. 35.6 A wide variety of street patterns can be generated by selecting the major and minor street patterns. Left: organic major and raster minor; Middle: raster major and raster minor; Right: radial major and organic minor

Fig. 35.7 Walls, streets, fences, and power-lines generated from rules executed on dynamic graph shapes

35.4 Writing CGA Rules for Fun and Profit

CityEngine rules are written in the Computer Generated Architecture (CGA) programming language. Writing a simple CGA rule can be quick and effortless; however, writing a realistic or flexible rule is an involved process. A library of existing rules is provided, and further rules can be found online. The fastest route to creating a 3D scene from a 2D map is by combining and parameterizing these existing rules, without ever writing CGA code ourselves.

Pre-installed rules can be found in the *ESRI.lib* project. A further selection of well-written rules for a variety of circumstances can also be found in the tutorials and

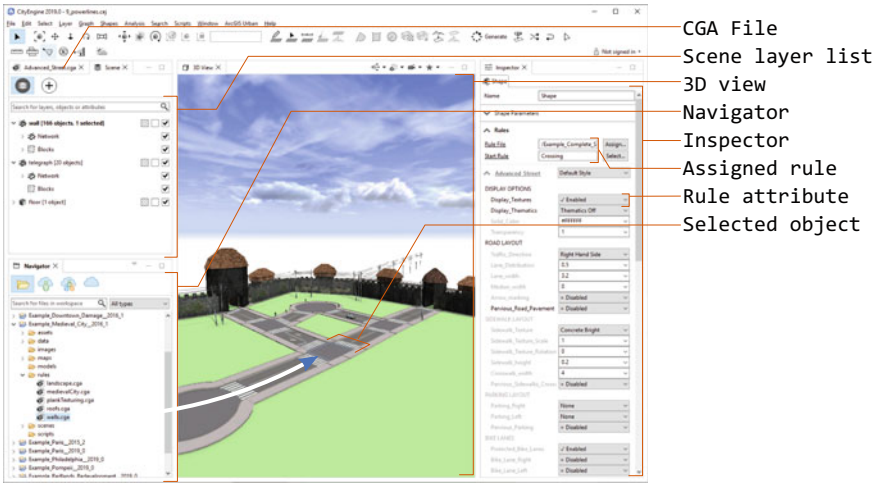


Fig. 35.8 CityEngine user interface elements. Orange: important elements of the interface. Blue: dragging a rule onto the selected shape to generate a 3D model

downloads dialog (*Help* → *Download Tutorials and Examples*). Finally, many user-generated rule packages (single .RPK files containing rules and resources) of varying quality can be found online ("ArcGIS content search" with keyword CityEngine; Esri 2019c). Exploring existing rules is a powerful way to understand how models can be generated using the CGA language. As rules can take a lot of time to write, reusing existing rules is advisable wherever possible; libraries should be used before writing CGA code ourselves.

To apply a rule or rule package, we may drag the rule package or file from the navigator onto a shape as shown in Fig. 35.8. By selecting a group of shapes before dragging, we may assign the rule to a number of shapes at once. The Inspector panel allows us to customize rules in a variety of ways. Various options exist for selecting shapes by layer or start rule can be found by right-clicking on a shape. After assigning a rule, there is a short delay while the rule is compiled and evaluated to create a model. If we desire more control, the Inspector contains more detailed options for the shape, including the CGA rule file, Start rule, and the previously mentioned rule attributes.

35.4.1 Writing Rules

While the mythos of "coders" and "software engineers" may have elevated programming to the status of a divine art, the reality is much more down to earth. CGA is a simpler language than the likes of Python, relying on a few basic operations which are repeatedly applied to write a rule. We find that undergraduate students are able to create their own rules after a few sessions with CityEngine. Those with experience of

complex languages such as C or C++ must learn the CGA way of doing things which is more *functional* than they are used to. The dialect of CGA used in CityEngine has evolved from the version presented in the initial academic publication (Müller et al. 2006); care must be taken when comparing rules from different versions.

We take the opportunity here to untangle the term "shape" in CityEngine. This has been overused to describe both the input shapes (described in the previous sections) and the shapes which are passed between rules in CGA. CityEngine refers to these intermediate shapes as "CGA shapes"; here, we will use the term geometry. This regrettable confusion is somewhat caused by the academic origin of CityEngine, where our input shapes did not exist.

A CGA rule file is a text document containing a collection of rules. A rule is analogous to a *function* or *method* in other programming languages. Each rule is identified by its name and set of parameters: X(1) is a different rule to X(1,2). As the rule is executed, it can call various operations, as well as other rules. Operations are analogous to *library functions* in other programming languages. As parent rules use operations to create new geometries, they label each with a child rule. If this rule exists, it will then be executed on the child geometry. Unlike the academic description of CGA (Müller et al. 2006), there is no concept of priority; rules are evaluated purely according to their parent rule.

Each rule transforms a piece of geometry into new geometries (or nothing); the result is a 3D mesh model consisting of all the geometry that cannot be further transformed. The initial geometry is the input shape to which the initial rule (sometimes designated with the @Startrule annotation) is applied. The rule also has access to attributes, which allows the rule behavior to be customized by the user or a data source. Attributes and parameters are used in the same way other programming languages use *variables* to customize behavior. Most of the attributes' values can be set and read by various operations. Attributes are sometimes taken as additional context for operations to define and refine behavior. For example, predominant orientation and origin information are encoded in the scope and pivot attributes. When the split operation is used in the y-direction, this direction is relative to this orientation given by the scope and pivot locations stored in attributes.

The typical pattern of programming in CGA is to repeatedly expand-then-divide geometry. The rule to create a building model may start with a lot shape, expand with an extrude operation to create prism geometry as high as the building, and then use a comp operation to divide the prism into various faces. The face pointing upward expands to create a roof with a roofGable operation, while side faces are divided using the split operation to become floors and then windows. Another extrude operation finally recesses the windows into the façade. We continue to study such operations in more detail.

35.4.1.1 Operations

Learning to write CGA rules is predominantly the process of learning the various operations and their effects on geometry and attributes. While the complexity of

existing rules can be overwhelming to the new user, the compact set of CGA operations presents a shallow learning curve.

CGA is a programming language designed to do one thing—model urban environments—and not much else. For this reason, we would describe it as a domain-specific (programming) language (DSL). For other domains, there are other programming languages: We may use L-Systems (Prusinkiewicz 1986) to generate flora or URDF (2019) to create robots. Because CGA is a DSL, its operations are carefully curated for the urban domain. A lot of theoretical effort was expended in finding a compact yet expressive set of operations. In contrast, general-purpose procedural modeling languages, such as Houdini (2019) and Rhino (2019), are not specialized in a single domain and have many complex operations to learn. Figure 35.9 introduces a handful of key CityEngine operations.

By repeatedly applying these operations, we can create a large variety of urban geometries. For example, the setback, extrude, comp, and roofGable operations can be used to create a house with a recessed top story and a gabled roof, as in the following Fig. 35.10.

An important observation is that CGA does not contain loop or repeat operations. To achieve repeating geometry (such as windows on a building façade or trees along a street), we can use the split operation with the asterisk (*) modifier to split a parent

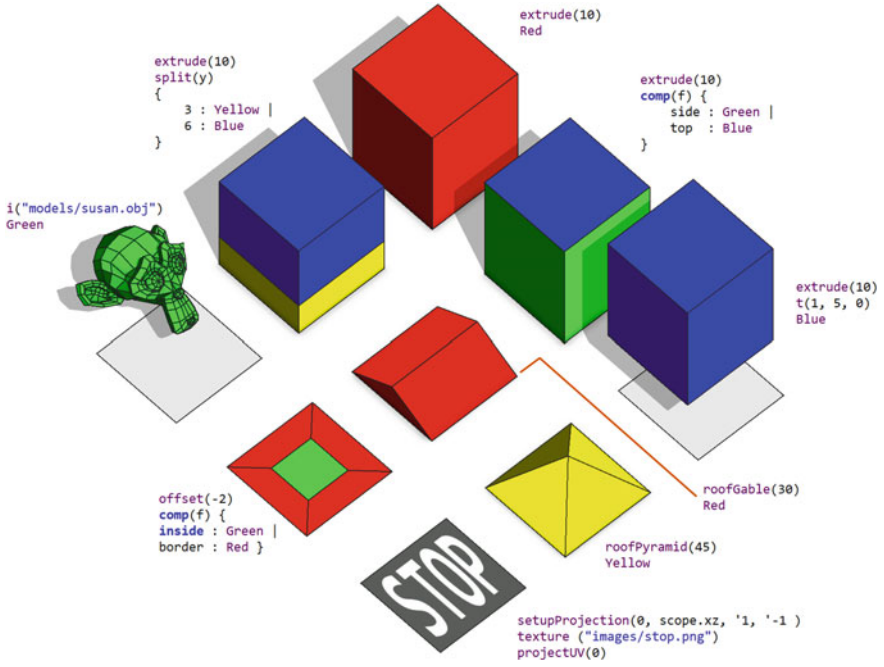
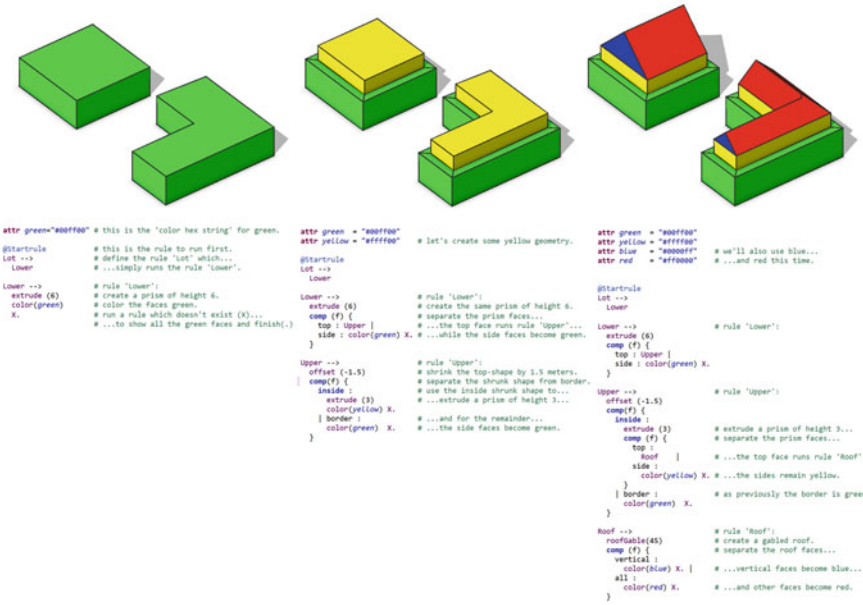


Fig. 35.9 CityEngine has over 60 operations. Here, we show a selection applied to a square input shape (gray), as well as example usage. Trivial rules with the names of colors (Red, Blue, etc.) are not shown, but would be included in the rule file



```
attr green="#00ff00" # this is the 'color hex string' for green.

@StartRule           # this is the rule to run first.
Lot -->              # define the rule 'Lot' which...
  Lower              # ...simply runs the rule 'Lower'.

Lower -->            # rule 'Lower':
  extrude (6)        # create a prism of height 6.
  color(green)       # color the faces green.
  X.                 # run a rule which doesn't exist (X)...
                     # ...to show all the green faces and finish(.)
```

```
attr green  = "#00ff00"
attr yellow = "#ffff00"   # let's create some yellow geometry.

@StartRule
Lot -->
  Lower

Lower -->                 # rule 'Lower':
  extrude (6)             # create the same prism of height 6.
  comp (f) {              # separate the prism faces...
    top : Upper |         # ...the top face runs rule 'Upper'...
    side : color(green) X. # ...while the side faces become green.
  }

Upper -->                 # rule 'Upper':
  offset (-1.5)           # shrink the top-shape by 1.5 meters.
  comp(f) {               # separate the shrunk shape from border.
    inside :              # use the inside shrunk shape to...
      extrude (3)         # ...extrude a prism of height 3...
      color(yellow) X.
    | border :            # ...and for the remainder...
      color(green)  X.    # ...the side faces become green.
  }
```

```
attr green  = "#00ff00"
attr yellow = "#ffff00"
attr blue   = "#0000ff"   # we'll also use blue...
attr red    = "#ff0000"   # ...and red this time.

@StartRule
Lot -->
  Lower

Lower -->                 # rule 'Lower':
  extrude (6)
  comp (f) {
    top : Upper |
    side : color(green) X.
  }

Upper -->                 # rule 'Upper':
  offset (-1.5)
  comp(f) {
    inside :
      extrude (3)         # extrude a prism of height 3...
      comp (f) {          # separate the prism faces...
        top :
          Roof    |       # ...the top face runs rule 'Roof'...
        side :
          color(yellow) X. # ...the sides remain yellow.
      }
    | border :            # as previously the border is green.
      color(green)  X.
  }

Roof -->                  # rule 'Roof':
  roofGable(45)           # create a gabled roof.
  comp (f) {              # separate the roof faces...
    vertical :
      color(blue) X. |    # ...vertical faces become blue...
    all :
      color(red) X.       # ...and other faces become red.
  }
```

Fig. 35.10 A progression of three CGA rule files using operations including extrude, comp, and roofGable, accompanying models shown above. Note how we start with a simple rule and gradually extend it to create more complex geometries following the expand-then-divide paradigm. The green text highlights comments which are ignored by CityEngine, but help humans to understand the code

shape into a repeating number of child shapes with the same rules. This is illustrated in Fig. 35.11.

In our final example, we create geometry for streets. To create highway lanes, we wish to split down the long axis of the streets, which may be curved. The UV variant of the split operation achieves this. Finally, we may wish to add texture maps (bitmap images) over our geometry instead of simple colors using the texture operations, as in Fig. 35.12.

35.4.2 Modeling Workflow

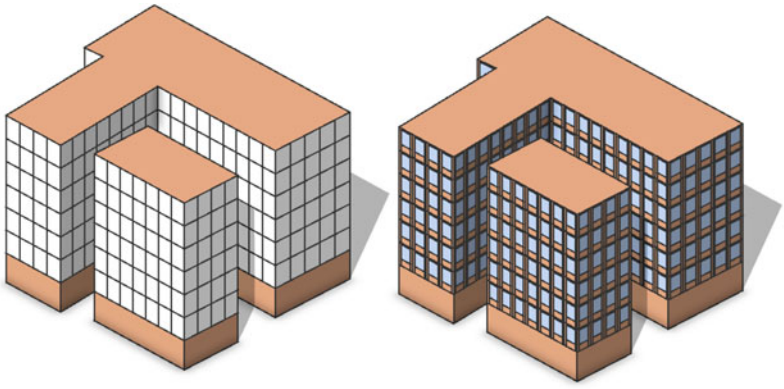
Creating larger rule files can be a daunting task for those new to writing code. This is a skill that requires time to practice and learn, but when a little knowledge is gained is often intoxicating:

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. (Brooks 1995)

This initial excitement often causes problems with inexperienced programmers; overconfidence causes a failure to understand the characteristics of a growing code

base. As many small problems in the code ("bugs") become entrenched, it can become very time-consuming to make even small changes. We can provide some general guidance and tools which can help us build large CGA programs:

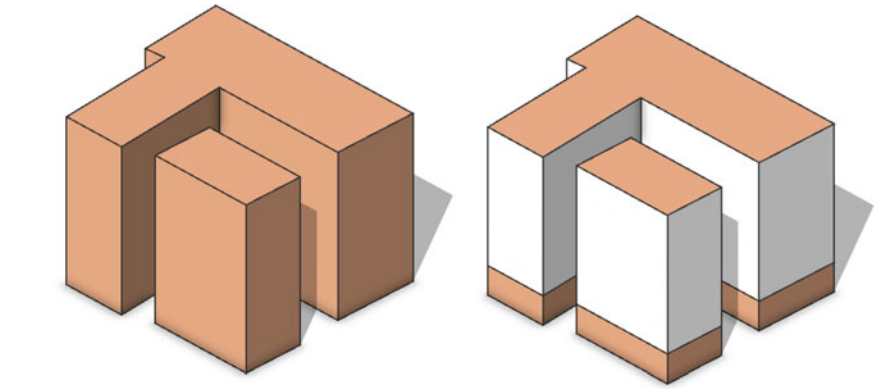
- Write small pieces of code at a time and test them frequently. This makes it much quicker to track down and isolate issues. If you cannot understand some behavior, it is frequently the case that too much code was written before trying to run it.
- Create reusable rules. A small rule that you have created which generates an "Acme brand window" may be reused if kept in separate file. CGA provides the import functionality to facilitate using this window rule in other rule files.
- Read the provided CGA documentation (*Help menu* → *CGA reference*).



```
@Startrule
Lot -->                 # rule 'Lot'
    extrude (20)
    comp(f) {
       side: Facade |
       top : Brick
    }

Brick -->               # rule 'Brick'
    color("#d29a78") X.

Facade -->              # rule 'Facade':
    split(y) {
      3.5: Brick |
      ~1 : TopFloors    # the remainder runs TopFloors.
    }

TopFloors -->           # rule 'TopFloors':
    split(y) {          # splits the face in the y direction...
      ~3: Floor         # aprox. 3m slices run the Floor rule.
    }*                  # the * causes the split to repeat.

Floor -->               # rule 'Floor':
    split(x) {          # split along x (sideways)...
      ~2: X.            # ...creating geometry aprox. 2m wide.
    }*                  # the * causes the split to repeat.
```

```
@Startrule
Lot -->                         # rule 'Lot':
    extrude (20)
    comp(f) {
       side: Facade |
       top : Brick
    }

Brick -->                       # rule 'Brick':
    color("#d29a78") X.

Facade -->                      # rule 'Facade':
    split(y) {
        3.5: Brick |
        ~1 : TopFloors
    }

TopFloors -->                   # rule 'TopFloors':
    split(y) {
      ~3: Floor
    }*

Floor -->                       # rule 'Floor':
    split(x) {
        ~2: Tile                # slices of the floor run Tile
    }*

Tile -->                        # rule 'Tile':
   split (x) {                  # split tile sideways (x)
      0.3 : Brick |             # left 30cm becomes brick
      ~1  : split(y) {          # split the remainder vertically
        0.8 : Brick |           # bottom 80cm becomes brick
        ~1 : color ("#afc6e9") # remainder is a blue 'window'
             X. |
        0.2 : Brick             # top 20cm becomes brick
      } |                       # finish off the first (x) split
      0.3 : Brick               # right 30cm becomes brick
   }
```

Fig. 35.11 Example of using the split rule to subdivide a façade to create windows



```
@Startrule
Lot -->                 # rule 'Lot':
    extrude (20)        # create a prism of height 20 meters.
    comp(f) {           # sepearate the faces of the prism...
       side: Brick |    # ...sides run the rule "Brick"...
       top : Brick      # ...as does the top.
    }

Brick -->               # rule 'Brick':
  color("#d29a78") X.   # color the shape, and display.
```

```
@Startrule
Lot -->                 # rule 'Lot':
    extrude (20)
    comp(f) {
       side: Facade |   # run the 'Facade' rule on the sides.
       top : Brick
    }

Brick -->               # rule 'Brick'
    color("#d29a78") X.

Facade -->              # rule 'Facade':
    split(y) {          # split the facade in the y (up)...
      3.5: Brick |      # ...the bottom 3.5 meters is brick...
      ~1 : X.           # ...the remainder is blank.
    }
```

Fig. 35.11 (continued)

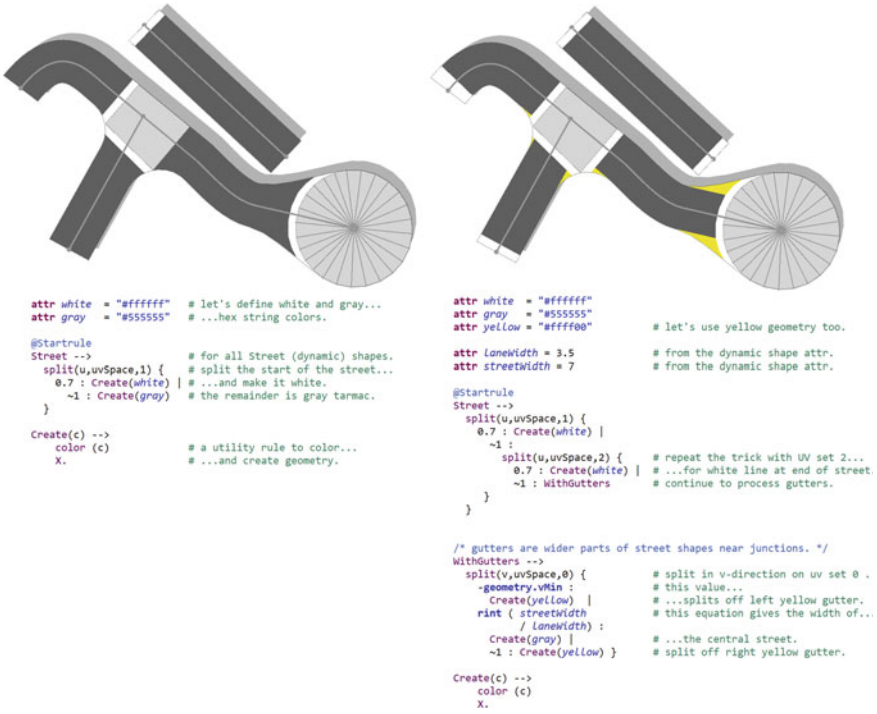
- It is easy to get lost in the details of programming and write code that is easy to understand today but difficult to understand in a week's time when you have forgotten the details. Use code comments (sections of code which the computer does not see) to keep notes for yourself and inform future readers. CityEngine comments can be created in two ways:

  ```
  //everything on this line is a comment
  /* everything between the two asterisks is a comment */
  ```

- Collections of rule files can be large, written by multiple people, have multiple versions, or can even evolve different branches as they are developed. For these reasons, programmers will typically use a version control system (such as the insensitively named git (git 2019)) to manage their code.
- Be aware of the keyboard shortcuts and context (right-click) menus available in CityEngine. For example, if you have a shape selected with a rule and are editing the rule in the text editor, *Ctrl + S* followed by *Ctrl + G* (on Windows or Linux; use the command key instead of Ctrl on OS X) will save and show the updated 3D shape. In the 3D view, the *F* key will move the view to show the selected object, or *F9–F12* will show and hide various classes of objects.

Beyond general programming etiquette, CityEngine provides several bespoke mechanisms to help writing CGA rules. The Model Hierarchy panel shows a graph of the different rule applications (*Window* → *Show Model Hierarchy,* Fig. 35.13). This shows the *Inspect Model* tool button, which can be used to select a building to analyze (Note that *Inspect Model* is a different piece of functionality to the Inspector panel.). The resulting graph is shown in the panel, with every rule application illustrated by a gray arrow. Lines connect parent/child rule pairs. By selecting a rule in the graph, the 3D view will highlight the resulting geometry and show the scope, pivot, and trim planes valid for the application of the rule. Right-clicking on a rule node in the graph gives the option to jump to the corresponding portion of CGA. A single CGA rule will typically be applied in different locations and so will appear multiple times in the graph.

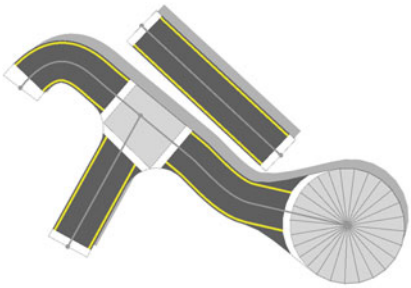
Another tool provided by CityEngine is the Façade Wizard (*Window* → *Show Façade Wizard*). For a single 2D façade, this aids in generating the split and extrude operations required for a well-parameterized façade.

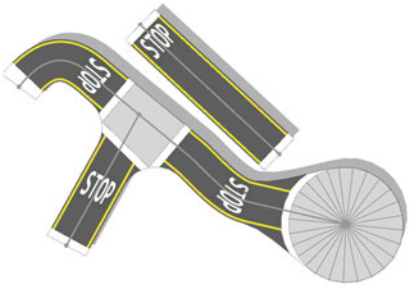


```
attr white  = "#ffffff"   # let's define white and gray...
attr gray   = "#555555"   # ...hex string colors.

@StartRule
Street -->                # for all Street (dynamic) shapes.
  split(u,uvSpace,1) {    # split the start of the street...
    0.7 : Create(white) | # ...and make it white.
     ~1 : Create(gray)    # the remainder is gray tarmac.
  }

Create(c) -->
    color (c)             # a utility rule to color...
    X.                    # ...and create geometry.
```

```
attr white  = "#ffffff"
attr gray   = "#555555"
attr yellow = "#ffff00"           # let's use yellow geometry too.

attr laneWidth = 3.5              # from the dynamic shape attr.
attr streetWidth = 7              # from the dynamic shape attr.

@StartRule
Street -->
  split(u,uvSpace,1) {
    0.7 : Create(white) |
     ~1 :
       split(u,uvSpace,2) {       # repeat the trick with UV set 2...
         0.7 : Create(white) |    # ...for white line at end of street.
         ~1 : WithGutters         # continue to process gutters.
     }
  }

/* gutters are wider parts of street shapes near junctions. */
WithGutters -->
  split(v,uvSpace,0) {            # split in v-direction on uv set 0 .
    -geometry.vMin :              # this value...
     Create(yellow)  |            # ...splits off left yellow gutter.
    rint ( streetWidth            # this equation gives the width of..
          / laneWidth) :
     Create(gray) |               # ...the central street.
     ~1 : Create(yellow) }        # split off right yellow gutter.

Create(c) -->
    color (c)
    X.
```

Fig. 35.12 Example of creating models for street shapes. The split rule is used with the UV parameter to split curved areas. The three different street UV sets split from different sides of the shapes. Finally, the normalize UV and texture commands create "stop" markings






```
attr white  = "#ffffff"
attr gray   = "#555555"
attr yellow = "#ffff00"

attr laneWidth = 3.5
attr streetWidth = 7

@Startrule
Street -->
  split(u,uvSpace,1) {
    0.7 : Create(white) |
      ~1 :
        split(u,uvSpace,2) {
          0.7 : Create(white) |
          ~1 : WithGutters
      }
  }

WithGutters -->
  split(v,uvSpace,0) {
    -geometry.vMin :        # on second thought, let's...
      Create(gray)  |       # ...make the gutters gray.
    rint ( streetWidth
          / laneWidth) :
      WithoutGutter |       # create non-gutter gometry.
      ~1 : Create(gray) }

WithoutGutter -->
  split(v,unitSpace,0) {    # split along street edges (v).
    0.4 : YellowLine |      # to create left yellow line...
    ~1 : Create(gray) |     # ...central area fills remainder.
    0.4 : YellowLine        # and right yellow line.
    }

YellowLine -->
  split(v,unitSpace,0) {    # split along street edge (v).
    0.1 : Create(gray) |    # 5cm of gray...
    ~1 : Create(yellow) |   # ...then a 10cm yellow strip...
    0.1 : Create(gray)      # ...and a final 5cm of gray.
}

Create(c) -->
    color (c)
    X.
```

```
attr white  = "#ffffff"
attr gray   = "#555555"
attr yellow = "#ffff00"

attr laneWidth = 3.5
attr streetWidth = 7

@Startrule
Street -->
  split(u,uvSpace,1) {
    0.7 : Create(white) |
      ~1 :
        split(u,uvSpace,2) {
          0.7 : Create(white) |
          ~1 : WithGutters
      }
  }

WithGutters -->
  split(v,uvSpace,0) {
    -geometry.vMin :
      Create(gray)  |
    rint ( streetWidth
          / laneWidth) :
      WithoutGutter |
      ~1 : Create(gray) }

WithoutGutter -->
  split(v,unitSpace,0) {
    0.4 : YellowLine |
    ~1 : WithoutStop |          # create stop markings.
    0.4 : YellowLine
    }

YellowLine -->
  split(v,unitSpace,0) {
    0.1 : Create(gray) |
    ~1  : Create(yellow) |
    0.1 : Create(gray)
}

WithoutStop -->
  split(u,uvSpace,0) {          # split from the start of the street.
    -geometry.uMin :            # any junction areas...
      Create(gray) |            # ...become gray.
    ~1 :                        # everything else...
     split(u,unitSpace,0) {     # ...is split again in units (meters)...
       3 : Stop |               # ...to create a stop sign.
       ~1 : Create(gray) }      # everything else is a regular street.
  }

Stop -->
  normalizeUV (0, uv,           # stretch the image over all..
    collectiveAllFaces )        # ...the current geometry.
  texture("images/stop.png")    # texture with the stop.png image.

Create(c) -->
    color (c)
    X.
```

Fig. 35.12 (continued)

To deliver a CityEngine rule to an end user in a convenient format, use a rule package. This can be built by selecting the CGA file to export in the navigator, right-clicking, and selecting *Share As*.... Additional resources and metadata are specified in the dialog box. In this way, the resulting .RPK file may include many individual CGA files and other resources such as data in text files and texture images. Such a package is easily distributed as a single file, and Esri provides a cloud system to distribute rules.

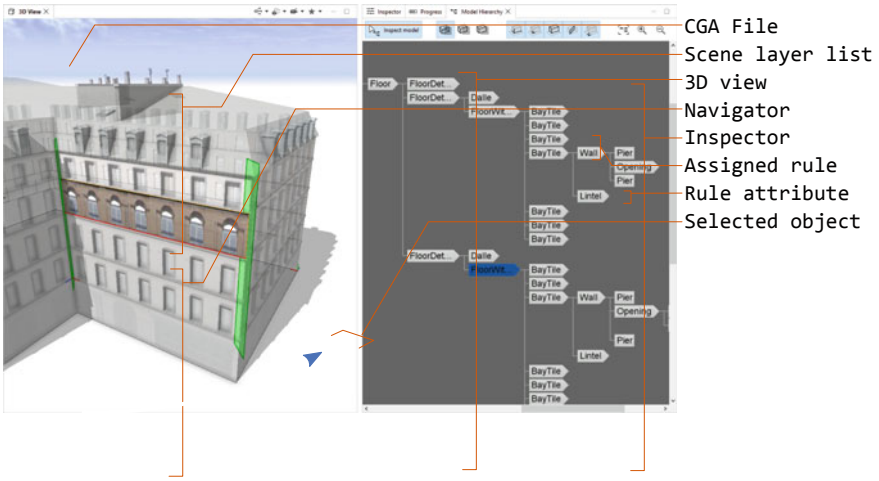


Fig. 35.13 The Model Hierarchy is a very useful tool for visualizing geometry. Left: a 3D view of a model from the first figure. The selected rule is highlighted and rendered with a solid color; the scope, pivot, and trim planes are also visualized. Right: the rule hierarchy identifies the rule which created the selected geometry. Clicking on another rule will show that rule's associated geometry. Note the Inspect Model button (top center) which is used to enable the Model Hierarchy functionality

35.4.3 Attributes

Having built our rules and assigned them to our shapes, we are often interested in further customizing the rule's expression using attributes.

Attributes are used to refine the evaluation of models within a rule application. They allow a rule to be generalized. For example, consider a number of otherwise identical buildings constructed from different materials; instead of a separate rule for each material, we may use a single rule with an attribute for the building material. Attributes can control any behavior of a rule, but typically, control features such as building height, age, or the number of pedestrians created on the sidewalks. CityEngine shows many of the available attributes for the selected shape and rule in the Inspector panel (Fig. 35.14); some rules have a great many attributes. The default attribute values are set by the rule. However, users can override the source of attributes to allow the rule to respond to different inputs.

The attributes in CityEngine have a multitude of different sources, and the interdependencies between them can be complex. Attribute sources include:

- Rule-sourced (Rule default), the default attribute behavior
- User-sourced
- Shape-sourced (Object attributes)
- Image- or shape-driven (Layer attributes).

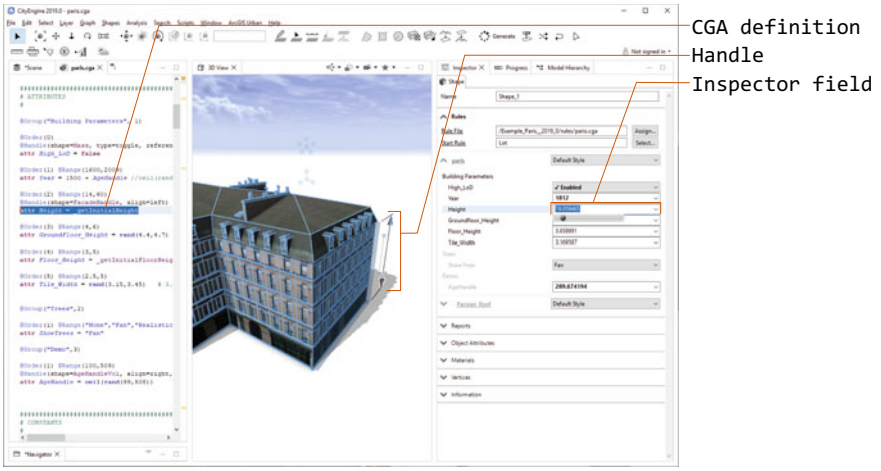


Fig. 35.14 Attributes are defined in the CGA file (left) and are edited either with handles (center) or using the Inspector (right)

These can be selected by clicking the down arrow next to an attribute in the Inspector panel and selecting *Connect Attribute*…. Rule-sourced attribute values are given in the CGA rule file. These attributes can be random; this feature can be used to add variation to a rule applied many times; for example, every building may be generated with the same rule, but given a height that is randomly selected between 10 and 20 m [attr height = rand (10,20)].

To allow users to change an attribute without editing the CGA file, attributes edited in the Inspector become user-sourced attributes. However, we may wish our attributes to come from other sources which may be driven by data. Object attributes are visible in the Inspector (under the *Object Attributes* heading) when a shape is selected. Object attributes can come from input data sources (e.g., OpenStreetMap data often gives every lot shape a building height attribute) or are created by dynamic shapes (e.g., the connectionStart and End attributes are added automatically to street shapes to specify the adjacent junction types).

Layer attributes sample their values from other shapes or a bitmap, as illustrated in Fig. 35.15. For example, we can drive the height-of-building attribute by using a georeferenced heightmap that has been captured by aerial LiDAR. In this way, we can control a rule using several different data sources. This approach significantly improves the accuracy of resulting geometry over a purely rule-driven procedural pipeline.

Finally, it is useful to know that the attributes for multiple shapes can be edited at once by selecting several shapes. Multiple shapes can be selected by shift-clicking or by dragging a selection box around them. Alternately, by right-clicking on shapes in the 3D view, various automatic selection options allow selection of many shapes within a layer. The Inspector shows the available attributes for the entire selection, and editing an attribute or source applies that attribute change to all the selected shapes.

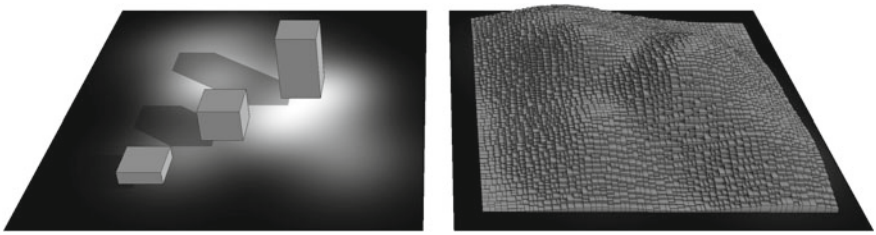


Fig. 35.15 Left: a black and white image imported as a texture is used to drive the height attribute of three rectangular suspended shapes, each with the same simple extrude rule. The white parts of the texture are sampled to large values, which are expressed as tall cuboids; black areas are small values which become short cuboids. Right: in this way, we may sample attributes from the same texture to vary building height (or any other attribute) across a city according to an image

35.4.4 Exploring Design Space

As a designer using CityEngine, the number of decisions that must be made can be very high. Complex rules present hundreds of attributes, and these must be aligned to user requirements, artistic visions, and practical considerations. Because every additional attribute adds a dimension to the design space, it can take a lot of time to explore large, heavily parameterized rules. Further, we may wish to design multiple scenarios: different rules, attributes, and shapes solving the same problem that we wish to compare side by side. CityEngine provides a Python interface for advanced programmers to control attributes (and many other scene elements) using custom code; typical uses are to create video animations of attributes or run custom design-space search algorithms. Most users, however, will want to avoid such complexities.

CityEngine presents a number of tools to help explore this design space of attributes visually. As we have seen, the simplest of these is the Inspector panel which arranges the attributes in groups specified by the rule file and allows the different attribute sources to be selected in a 2D interface. Given the large number of attributes in a rule such as the Paris example, it is often useful to see a visual representation of those attributes next to the 3D model. Handles present this functionality by showing the attributes (such as height) as controls in the 3D view. The handle system was inspired by the dimension lines of engineering diagrams, as introduced by Kelly et al. (2015). When a model with handle functionality is selected in the 3D view, the handles are shown at the edges of the model depending on the viewpoint. Various handles control different types of values: Boolean toggles, multiple-choice dials, distance-as-value dimension lines, and color selector triangular handles are available. The handle locations, behavior as the viewpoint moves, and appearance are defined by the @Handle annotation in the CGA rule file. They are designed by the rule creator and are only available if the rule author chooses to use them. Often the rule author will choose to expose only the most-used attributes using handles to avoid overcrowding the screen.

Handles change the value of an attribute throughout an entire rule evaluation for a single shape. There are situations where we wish to edit an attribute within a rule evaluation, for example, to make one story of a building taller than the others or to move the location of a single window in a large façade. In this situation, we can use local edits. These allow us to edit attributes with handles. Local edits are created by selecting the *Local Edits Tool*; depending on how the rule is structured, this tool may allow us to edit all local attributes in a row, column, or more complex patterns at once. Local edits are discussed further by Lipp et al. (2019).

As we modify rule attributes, we may be trying to achieve an objective target such as a target floor area for a building or group of buildings. CityEngine's reporting mechanism allows rules to collate such information and then prepare a summary report for each model. The report operation accumulates values whenever it is invoked, returning a sum total for the entire model [we may use the operation report ("area", 200)]. Multiple values (floor area, room volume, etc.) can be accumulated for each rule and displayed in the Inspector as a table. If CityEngine's dashboard functionality is used, these tables can be presented as a range of graphs which update automatically. They can show results over all models in the scene or only those selected.

By taking the time to add reports to your models and using the dashboard functionality, it becomes possible to explore the design space interactively with a wide range of users. For example, clients may appreciate being able to use the handles to edit building heights and receive instant feedback on the effects of available floor area and construction costs.

Beyond raw reported analytics, we may be interested in the visual consequences of our designs. CityEngine provides a range of tools for measuring distance and area in the 3D scene (Fig. 35.16), but most interestingly provides visibility calculations; this highlights the areas of models which are visible or not from a certain location under a given field of view.

Finally, scenarios allow us to compare different events. Each scenario can contain different layers of content on top of a shared background. For example, three different developments proposed for a city block with different height can be shown, while the surrounding city remains constant. A scenario can be duplicated and edited to explore a new design space.

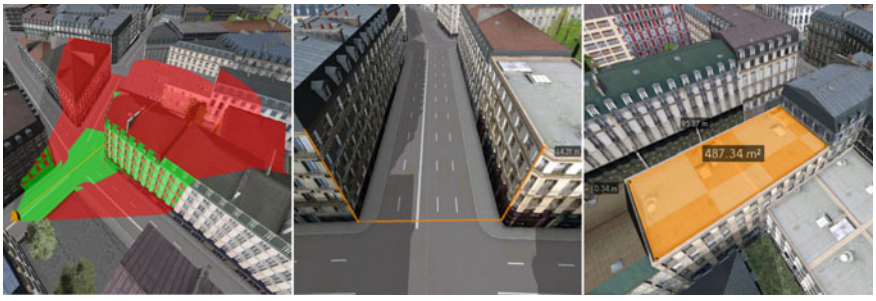


Fig. 35.16 Analysis tools. Left: viewshed calculations showing visible (green) and occluded (red) areas. Middle: path length measuring tool. Right: area measuring tool

35.5 Beyond CityEngine: Export Pathways

After we have painstakingly created shapes, written rules, and adjusted parameters to generate our 3D reconstruction, we will want to view, export, and share our CityEngine scenes.

It should be noted that CityEngine's 3D view can create images with a reasonable-quality lighting model. There are options in the viewport panel (*View Settings*) to enable shadows (as cast by the sun), ambient occlusion (more accurate shadows in geometry creases), and field of view (the angle of the scene we see). Images can be saved from the 3D (*Bookmarks* → *Save Snapshot…*).

CityEngine's 3D view renderer is a real-time OpenGL renderer similar to those used for video games. If we would like more accurate physically based rendering (PBR) and are prepared to wait for each image to render, we can use a third-party renderer (such as POV-Ray, LuxRenderer, Unity game engine, Autodesk 3ds Max, or Blender) to create accurate images. These renderers are complex pieces of software in themselves, and the mechanics and artistry of setting up lighting and materials to create beautiful photorealistic images are beyond this chapter. However, in Fig. 35.17, we compare the default CityEngine rendering to the physically based Cycles renderer in Blender. We note the high quality of light simulation (reflections, shadows, and color bleeding) and material appearance.

To use an external renderer, we must export our models as 3D meshes from CityEngine to another package. CityEngine offers a variety of different formats to export models (*File* → *Export Models*): Wavefront's OBJ is a commonly used interchange format, but other more exotic formats include Collada, Autodesk FBX, and Alembic. Then a typical pipeline in a 3D modeling application such as Blender is to import the 3D meshes, set up textures, and position the camera and lights. Finally, a render operation is performed that might take minutes or even days to produce a large high-quality image.

To share our finished 3D meshes online with others as 3D objects, rather than 2D images, there are several options. There is a rapidly growing selection of Web-based 3D hosts (Sketchfab, SketchUp 3D Warehouse, or Google's Poly) who will host OBJ meshes online so that they may be viewed in a browser. Links to the resulting Web pages can be shared with clients and colleagues. However, these general 3D sites lack support for many details from a CityEngine scene. Esri provides two solutions to this problem: the CityEngine Web scene exporter (*File* → *Export Models…*) and the separate application ArcGIS Urban (*ArcGIS Urban* → *Synchronize all scenarios*). This ensures that details such as lighting information, different scenarios, and shape information remain visible and interactive for viewers, although editing attributes is not supported. Esri provides a convenient pipeline from CityEngine to host Web scenes on their online platform; this includes support for a "split-screen" to show two scenarios side by side in the browser.

Immersive technologies are a recent and popular trend in 3D visualization. Virtual reality (VR) is the most popular medium: Users wear a headset (such as the Oculus Rift or HTC Vive) which tracks head motions and shows different images to each

Fig. 35.17 Top: CityEngine's default OpenGL real-time renderer without ambient occlusion or shadows. Middle: with ambient occlusion and shadows. Bottom: Blender's Cycles renderer takes 12 min to render this image with soft shadows and reflective glass. The mesh was exported to Blender in the OBJ format

eye to create a realistic and immersive 3D experience. Creating these experiences is still a technical process and requires the use of a video game engine; the most developed CityEngine pipeline uses the Unreal Engine. CityEngine 2019.0 includes a beta Unreal Engine model exporter, the output of which can be imported into Unreal via the Datasmith toolkit. The technical details are documented online and are likely to change in the near future (Esri 2019d).

The CityEngine VR experience presents a tabletop containing the models (Fig. 35.18). This presents the exported models on a tabletop in a virtual office. Users are able to explore the models by dragging the model on the tabletop. Optionally, the user can teleport to pre-designated sites in the 3D world to get a street-level view of the model. These design decisions avoid some of the discomfort of moving users through VR at high speeds. The tabletop interface eliminates motion sickness by allowing users to stand over the scene and explore it from a "virtually static" location.

There are downsides to VR as a presentation format. A minority of people still experience motion sickness or discomfort, the headsets are not suitable to be worn for long periods of time, and they are still low resolution when compared to desktop monitors. These limitations are rapidly diminishing as improved hardware and software interfaces become available. However, for applications where immediate impact or immersion is important, they can be very powerful tools for stimulating discussion and gauging impact.

Fig. 35.18 CityEngine virtual reality presents a tabletop model to navigate using the controllers (right). Multiple users are supported (second user's headset shown top center)

35.6 Conclusion

CityEngine provides several pieces of unique functionality to the urban designer's toolkit. The ability to work with rules, rather than concrete manual models, can massively reduce the time, increase the scale, and lead to a multitude of new workflows for designing urban spaces. These new workflows allow us to quickly iterate solutions in a "client's office" situation; the solutions can be visualized and quantitatively analyzed on-the-fly. Such innovations allow faster user feedback as well as a better understanding of the problem and solution spaces.

All new workflows come with caveats and CityEngine is no exception. When a non-programmer (who does not write rules) uses CityEngine, he or she faces a limited selection of rule files. A programmer will usually have to invest substantial time learning CGA and creating rule files appropriate to the problem. However, there are substantial resources available to aid both groups of users: Large libraries of rules are available online, and comprehensive API documentation is provided for the programmer.

CityEngine originally grew out of Pascal Müller's academic work at ETH Zürich (Müller 2010). The continuing development of the CityEngine software product has been quietly shadowed by academic works detailing the future innovations in the system (Schwarz and Müller 2015); such technologies and features often flow between other Esri products and CityEngine itself. Recent innovations in dashboard data presentation and pipelines for virtual realities reflect the exciting ongoing development of the system at Esri R&D Center Zürich.

References

Blender (2019) https://www.blender.org/. Accessed 30 July 2019
Brooks FP (1995) The mythical man-month: essays on software engineering. Anniversary Edition, 2/E. Pearson Education India
Esri (2019a) https://doc.arcgis.com/en/cityengine/latest/tutorials/introduction-to-the-cityengine-tutorials.htm. Accessed 30 July 2019
Esri (2019b) https://doc.arcgis.com/en/cityengine/. Accessed 30 July 2019
Esri (2019c) http://www.arcgis.com/home/search.html. Accessed 30 July 2019
Esri (2019d) https://community.esri.com/docs/DOC-11563-cityengine-vr-experience-for-unreal-studio. Accessed 30 July 2019
git (2019) https://git-scm.com/. Accessed 30 July 2019
Houdini (2019) https://www.sidefx.com/products/houdini/. Accessed 30 July 2019
Kelly T, Wonka P, Müller P (2015) Interactive dimensioning of parametric models. Comput Graphics Forum 34(2):117–129
Lipp M, Wonka P, Müller P (2014) PushPull++. ACM Trans Graphics (TOG) 33(4):130
Lipp M, Specht M, Lau C, Wonka P, Müller P (2019) Local editing of procedural models. Comput Graphics Forum 38(2):13–25
Maya (2019) https://www.autodesk.co.uk/products/maya/overview. Accessed 30 July 2019
Müller P (2010) Procedural modeling of buildings. Thesis, University of Zürich. https://doi.org/10.3929/ethz-a-006397747

Müller P, Wonka P, Haegler S, Ulmer A, Van Gool L (2006) Procedural modeling of buildings. ACM Trans Graph 25(3):614–623
Parish YI, Müller P (2001) Procedural modeling of cities. In: Proceedings of the 28th annual conference on computer graphics and interactive techniques. ACM, pp 301–308
Prusinkiewicz P (1986) Graphical applications of L-systems. In: Proceedings of graphics interface and vision interface '86, pp 247–253
Prusinkiewicz P, Lindenmayer A (2012) The algorithmic beauty of plants. Springer Science & Business Media
Rhino (2019) https://www.rhino3d.com/. Accessed 30 July 2019
Schwarz M, Müller P (2015) Advanced procedural modeling of architecture. ACM Trans Graph 34(4):107
SketchUp (2019) https://www.sketchup.com/. Accessed 30 July 2019
SpeedTree (2019) https://store.speedtree.com/. Accessed 30 July 2019
URDF (2019) http://wiki.ros.org/urdf. Accessed 30 July 2019
Vanegas CA, Kelly T, Weber B, Halatsch J, Aliaga DG, Müller P (2012) Procedural generation of parcels in urban modeling. Comput Graphics Forum 31(2pt3):681–690
Wikipedia (2019) https://en.wikipedia.org/wiki/Grome. Accessed 30 July 2019

Tom Kelly is a member of faculty at the University of Leeds, where he conducts computer graphics research and teaches user interfaces. Previously, he worked as a software engineer at Esri and as a video game developer.