

Procedural Modeling of Buildings for Simulation-Based Urban Synthesis



Urban modeling, or modeling of cities using a computer, is a rapidly developing and highly promising field. The ability to view and manipulate the structure of an entire city promises to be an invaluable tool in the fields of entertainment, training, and urban planning. However, urban modeling is not simple, and as a result, *manual* urban modeling can be very time-consuming. In particular, modeling the individual buildings within a city by hand is extremely tedious if any sort of variety is desired. The Computer Graphics and Visualization Lab’s Urban Synthesis project addresses the overall issues with urban modeling, and my work with procedural modeling targets the specific issue of creating believable, varied buildings. This abstract gives an overview of the Urban Synthesis project, followed by an explanation of my contribution.

Previous work in the field of urban modeling includes several projects by the CGVlab relating to the manipulation and visualization of urban layouts, one of which specifically uses urban simulation data to infer new content for part of a model. Also of particular interest to this project are the work done by Parish and Müller (“Procedural Modeling of Cities,” SIGGRAPH 2001); and Müller, Wonka, Haegler, Ulmer, and Van Goul (“Procedural Modeling of Buildings,” SIGGRAPH 2006). Both of these papers describe the application of *shape grammars* to different aspects of urban modeling, and CityEngine—a product which uses shape grammars for

generating building models— is being used heavily in my part of the CGVlab Urban Synthesis project.

The goal of this project is to generate realistic city models quickly and with a minimum of user interaction. This is critical for almost any usage of urban modeling, since users are more interested in applying their model than creating it. In order to make this possible, our project exploits a key observation: That there is an interaction between the geometric and socioeconomic variables associated with a city. For example, changing the elevation of the terrain in an area affects the area's accessibility, which in turn affects its land value, the relative wealth of households in the area, and finally the style of buildings. In this case, the geometric variable of elevation interacts with the socioeconomic variables of accessibility, land value, and household wealth, which then interacts with the geometric variable of buildings style. There are many such interactions, which leads to the Urban Synthesis project's approach: First, to create models which efficiently reproduce these interactions, and second, to provide a user-friendly interface for specifying the input to these models.

The process followed by our project is as follows: First, the user provides input in the form of initial values for several key geometric and socioeconomic variables. For example, the



user can manipulate the terrain, population density, and distribution of jobs. Next, an urban simulation is run in which the interactions between these variables are played out. After that, the data from the simulation is used to generate realistic road networks, land parcels, and building

locations. In addition, certain attributes for each building can be inferred from the simulation data. These include the building type (e.g. residential or commercial) and number of stories. Once the building locations and attributes have been defined, CityEngine is used to generate the actual geometry of the buildings. Finally, the terrain, road networks, and building models can be exported to rendering software to produce a photorealistic final output.

The purpose of my contribution is to generate buildings models for a synthesized city which are believable and which fit the characteristics of the urban model. In practical terms, this means I created the procedure for using CityEngine to generate building geometry from the location and attribute data as described above. Primarily, this revolves around the understanding and application of the CGA shape grammar.

A shape grammar is an application of the concept of mathematical grammars to the generation of geometry, and like all mathematical grammars, a shape grammar consists of symbols and rules. However, in a shape grammar the symbols correspond to shapes, and the rules correspond to operations on those shapes. Thus, the initial input to the shape grammar system is a *start symbol* along with a corresponding *initial shape*. In order to make the system more flexible, CGA also supports the use of *attributes*, which act as constants that can be used in the grammatical rules. These rules then replace the initial shape with other shapes, which are then replaced with other shapes until the desired result is achieved. Some of the replacement operations supported in CGA are *insert*, which simply replaces one shape with another, *split*, which splits a shape along a plane, and primitive transformations such as translate, scale, and rotate. Additional operations allow for the specification of material properties such as colors and textures. When the replacement process is finished (there are no more nonterminal symbols to replace), the resulting geometry and texture mappings are output.

Once the shape grammar has been created and suitable textures have been found (a good selection of building textures can be found on the Web), the pipeline for my part of the project is simple. First, data is imported in the form of parcel boundaries (which correspond to initial shapes) and their corresponding start symbols (which specify the type of building to create on each parcel). This data is stored in the standard OBJ format and is output by the main Urban Synthesis program. Currently CityEngine does not allow attribute values to be specified in an OBJ file, but it does allow the creation of *attribute maps*, which are image files that encode attribute values using color channels. Thus, for our project, the Urban Synthesis program creates an orthographic, top-down rendering of the city with parcels colored according to the desired attribute values. Then the CGA grammar is applied to each initial shape, beginning with its start symbol and incorporating the attribute values from the

```

House(floors, doorSideIndex) -->
  extrude(floors * floorHeight)
  comp(f) {
    doorSideIndex: HouseFacade("front") |
    side: HouseFacade("side") |
    top: HouseRoof
  }

HouseFacade(side) -->
  setupUV(0, 1.0, 1.0)
  set(material.colormap, wallTexture)
  split(y) {
    floorHeight: HouseFloor(side, true) |
    { ~floorHeight: HouseFloor(side, false) } *
  }

HouseFloor(side, isGroundFloor) -->
  case isGroundFloor && (side == "front"):
    split(x) {
      ~1: HouseWall |
      doorWidth: split(y) {
        doorHeight: HouseDoor |
        ~1: HousePlainWall
      } |
      ~1: HouseWall
    }
  else:
    HouseWall

HouseWall -->
  split(y) {
    doorHeight - windowHeight: HousePlainWall |
    windowHeight: split(x) {
      ~windowSpacing: HousePlainWall |
      {
        windowHeight: HouseWindow |
        ~windowSpacing: HousePlainWall
      } *
    } |
    ~1: HousePlainWall
  }

HouseRoof -->
  case geometry.isRectangular(1.0):
    alignScopeToGeometry(yUp, 0, 0)
    HouseGableRoof
  else:
    HousePyramidRoof

HouseGableRoof -->
  case scope.sx > scope.sz:
    roof("gable", houseRoofAngle, 1)
    comp(f) {
      bottom: NIL |
      aslant: HouseRoofTop |
      top: HouseRoofTop |
      side: HouseRoofSide
    }
  else:
    roof("gable", houseRoofAngle, 0)
    comp(f) {
      bottom: NIL |
      aslant: HouseRoofTop |
      top: HouseRoofTop |
      side: HouseRoofSide
    }

HousePyramidRoof -->
  roof("pyramid", houseRoofAngle)
  comp(f) {
    bottom: NIL |
  }

```

map. Finally, the resulting building models, along with their texture mappings, are exported back to the OBJ format for import into the rendering software.

This approach to generating buildings requires much less effort than creating models by hand, for several reasons. First, it allows the structure of buildings to be described semantically and lets the computer perform the difficult task of creating the actual polygons that make up the model. Second, it makes it very easy to re-use parts of buildings; for example, all four of the “house” models I created share the same “wall” and “front door” rules, which are written to adapt to any size and position. Thirdly, it allows great diversity to be achieved with a minimum of effort. For example, any building type can be placed on any size parcel and assigned any number of floors, and the result will be a believable model.

Grammar-based procedural modeling of buildings is thus a great advantage when speed and minimal input are the primary goals, and these are exactly the goals of the Urban Synthesis project. By following the steps outlined in this paper, we are able to quickly model enormous cities in great detail, which we believe will be useful to movie producers, game designers, and urban planners everywhere.

