Tutorial 7: Facade modeling

Table of Contents

Tutorial 7: Facade modeling

In this tutorial

Download items

- Tutorial data
- Tutorial PDF

Model the facade structure

This tutorial shows how to model a building from a picture and introduces some more complex CGA techniques. In this section, you'll create the basic structure of the facade with CGA rules.

Tutorial setup

Steps:

1. Import the Tutorial_07_Facade_Modeling project into your CityEngine workspace.
2. Open the `Tutorial_07_Facade_Modeling/scenes/FacadeModeling_01.cej` scene.

Facade modeling

This tutorial explains how to write a set of CGA rules to recreate a facade from a real-world photograph. The facade you're going to model is shown in the following image:

You'll continuously analyze the photograph in more detail as you proceed extending the rule set. You'll learn how to analyze an existing real-world facade and transfer its structure into CGA grammar rules. You'll also learn how premodeled assets can be used in CGA rules.

Create the rule file

Steps:

1. Click **New** > **CityEngine** > **CGA Grammar File**.
2. Make sure the container is set correctly (`Tutorial_07_Facade_Modeling/rules`), name the file `facade_01.cga`, and click **Finish**.

   A new CGA file is created, and the CGA Editor is opened. It's empty except for some header information (/* .... */) and the version tag version "2011.1".

Volume and facade

The actual creation of the building starts now. First, the mass model is created with the extrude operation. You'll use an attribute for the building height.

Steps:

1. Add the attribute height to the beginning of the rule file.

   ```
   attr height                        = 24
   ```

2. Write the starting rule using the extrude command, and call the shape Building.

   ```
   Lot --> extrude(height) Building
   ```

3. For this example you're only interested in the facade, so you'll use the component split to remove all faces except the front face, and then call the Frontfacade rule.

   ```
   Building --> comp(f) { front : Frontfacade}
   ```

   Attributes can have optional annotations such as @Group or @Range, which control the display of the attributes in the Inspector. See the CGA Reference for details on CGA annotations.

Floors

The front facade is split horizontally into the different floors, each with a floor_height attribute. The Floor shape is parameterized with the split.index, which is the floor index. This parameter will be passed to subrules to determine what elements to create on specific floors.

Steps:

1. For the top floor, the floorindex is set to 999. This allows you to identify this floor easily.

   Note the repeating split for the mid floors. This allows the building to adapt dynamically to different building heights, and fill the remaining vertical space with mid floors.

2. Define some attributes for the floor dimensions.

```
attr groundfloor_height         = 5.5
attr floor_height                 = 4.5
```

3. Add the rule.

```
Frontfacade -->
        split(y){ groundfloor_height    : Floor(split.index)         // Groundfloor
                | floor_height : Floor(split.index)                   // First Floor
                | floor_height : Floor(split.index)                         // Second Floor
                | {~floor_height : Floor(split.index)}*                 // Mid Floors
                | floor_height : Floor(999)                         // Top Floor, indexed
                                                        // with 999
                | 0.5 : s('1,'1,0.3) LedgeAsset}                    // The top ledge just
                                                        // below the roof
```

   Now you'll generate the facade for the first time.

4. Select the lot in the 3D viewport.
5. Assign the rule file by clicking **Shapes** > **Assign Rule File**, select your the `facade_01.cga` rule file and click **OK**.
6. Click the **generate** button on the top toolbar (or press Ctrl-G).

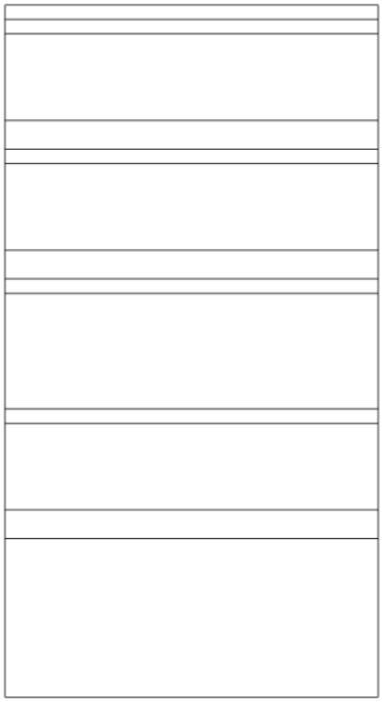
   The result looks similar to the following image:

Floor ledges

Floors are now split into ledge and tile shapes. Bottom ledges are specific to floors, so you'll use the floorindex parameter for this rule.

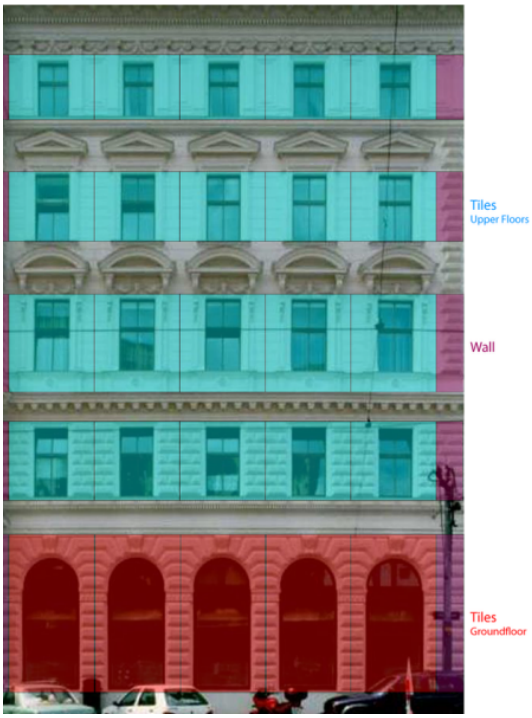
- The ground floor (floorindex 0) has no ledges and therefore calls Tiles only.
- Because windows start at floor level, there is no bottom ledge for the second floor. The balcony on this floor will be created in a later step.
- All other floors have bottom ledge, tile, and top ledge areas.

```
Floor(floorindex) -->
        case floorindex == 0 :
                Subfloor(floorindex)
        case floorindex == 2 :
                split(y){~1 : Subfloor(floorindex) | 0.5 : TopLedge}
        else :
                split(y){1 : BottomLedge(floorindex)
                  | ~1 : Subfloor(floorindex) | 0.5 : TopLedge}
```



Subfloor

Subfloors consist of small wall areas on left and right edges and repeating tiles in between.



Steps:

1. Add the following attribute on top:

```
attr tile_width                = 3.1
```

2. Next, add the rule:

```
Subfloor(floorindex) -->
        split(x){ 0.5 : Wall(1)
                        | { ~tile_width : Tile(floorindex) }*
                        | 0.5 : Wall(1) }
```

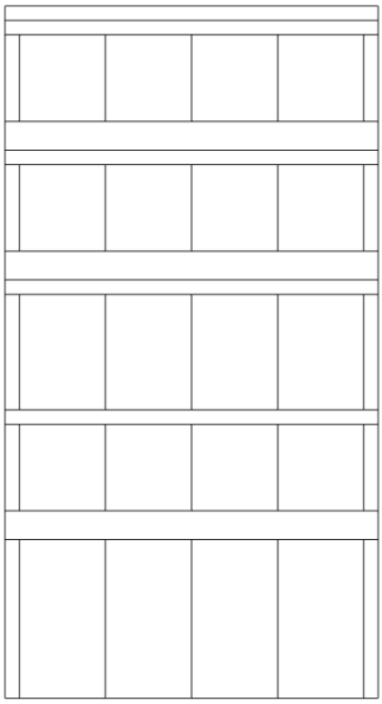
At this point, a parameterized Wall shape is added. This is important for a later step when you'll texture the facade. Looking at the facade photograph, you'll notice three wall types:

- Dark bricks with dirt texture
- Bright bricks with dirt texture
- Dirt texture only. This is mainly needed for facade assets that do not have a brick structure.

As you see from the following rule, the wall styles produce identical output. In a later step, you'll add different textures to the wall types.

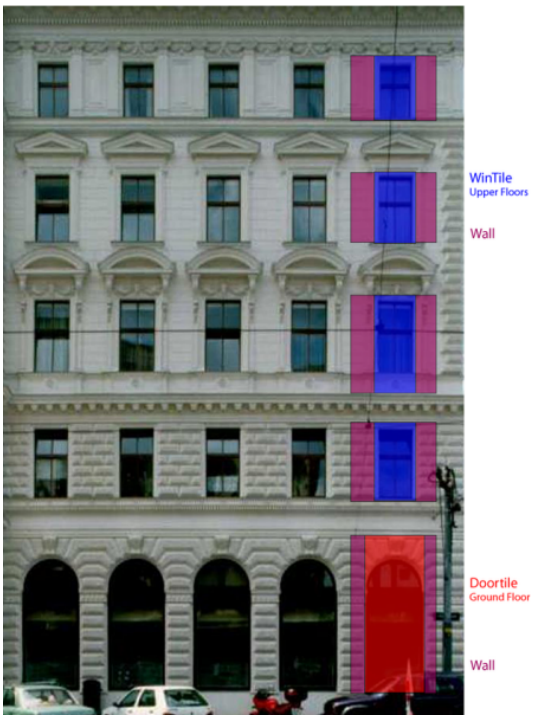
```
attr wallColor          = "#ffffff"
```

```
Wall(walltype) -->
        // dark bricks with dirt
        case walltype == 1 :
                color(wallColor)
        // bright bricks with dirt
        case walltype == 2 :
                color(wallColor)
        // dirt only
        else :
                color(wallColor)
```



Tile

Tiles are homogeneous on this facade. You only need to differentiate between the ground floor tiles and the upper floors.



Use the door_width and window_width attributes to set the different split dimensions.

```
attr door_width                = 2.1
attr window_width                = 1.4

Tile(floorindex) -->
        case floorindex == 0 :
                split(x){ ~1 : SolidWall
                        |  door_width : DoorTile
                        | ~1 : SolidWall }
        else :
                split(x){ ~1 : Wall(getWalltype(floorindex))
                        |  window_width : WindowTile(floorindex)
                        | ~1 : Wall(getWalltype(floorindex)) }
```

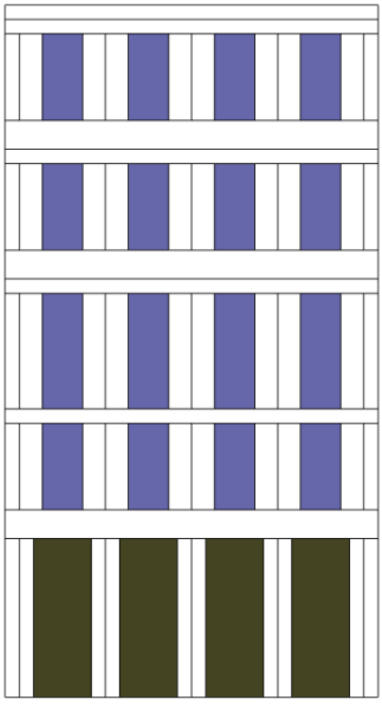
For the ground floor tiles, a new SolidWall shape was added. This is necessary because doors on the ground floor are inset from the facade. To avoid holes between door and wall, you'll use a solid wall element there by inserting a cube with a defined thickness. Because you'll use this thickness again in a later Door rule, you define it as a const variable wall_inset. Declaring values that are used more than once is a good idea, as it ensures that the same value is used in different rules.

```
const wall_inset = 0.4

SolidWall -->
                s('1,'1,wall_inset) t(0,0,-wall_inset)
                i("builtin:cube:notex")
                Wall(1)
```

Declare a function to get the wall type from the floor index. Looking at the facade, you see that there are dark textures on the ground and first floor, and bright textures on the others. The getWalltype function maps the floor index to the corresponding wall type.

```
getWalltype(floorindex) =
        case floorindex == 0 : 1
        case floorindex == 1 : 1
        else : 2
```



Now you'll learn how to use assets on this facade.

Insert facade assets

This section of the tutorial describes how to use premodeled assets on the facade.

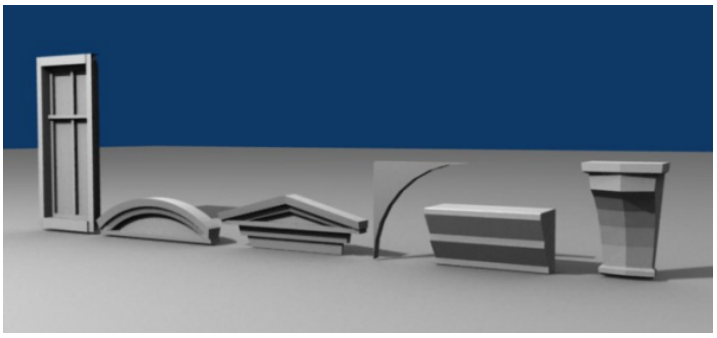
Steps:

1. Open the `Tutorial_07_Facade_Modeling/scenes/FacadeModeling_02.cej` scene file if it's not already open.
2. Open the `Tutorial_07_Facade_Modeling/rules/facade_02.cga` file.

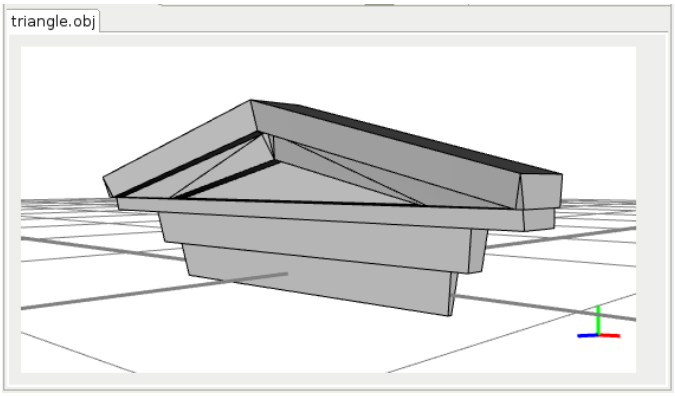
Assets

Looking at the photograph of the facade you're going to model, you see you need the following assets:

- Window—Used for the window elements
- Round Windowtop—Used for the ornaments above windows
- Triangle Windowtop—Used for the ornaments above windows
- Half arc—Used for arcs on the ground floor
- Ledge—Used for all ledges
- Modillion—Used for window ornaments and arc ornaments on the ground floor



These assets are already present in the assets folder of the tutorial project. You can preview these assets in the CityEngine Inspector by selecting the desired asset in the Navigator.



Asset declarations

It's a good idea to have all the asset declarations in the same place. Add the following lines below the attribute declarations to your rule file:

```
const window_asset                  = "facades/elem.window.frame.obj"
const round_wintop_asset          = "facades/round_windowtop.obj"
const tri_wintop_asset                 = "facades/triangle_windowtop.obj"
const halfarc_asset                 = "facades/arc_thin.obj"
const ledge_asset                 =
                          "facades/ledge.03.twopart_lessprojection.obj"
const modillion_asset                 =
            "facades/ledge_modillion.03.for_cornice_ledge_closed.lod0.obj"
```

Window

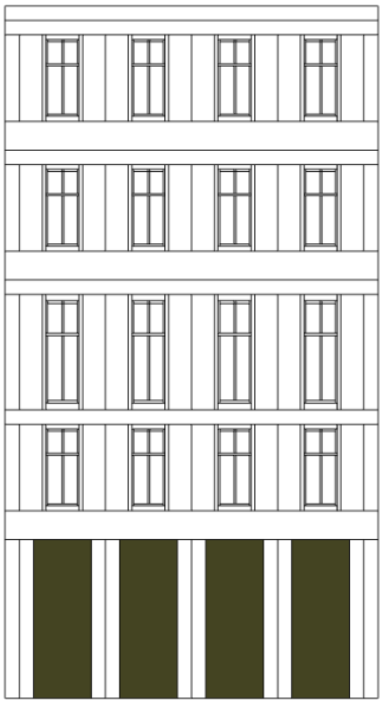
Steps:

1. You have the rules ready for the exact placement of the window assets. Call the Window shape in the WindowTile rule.

   ```
   WindowTile(floorindex) --> Window
   ```

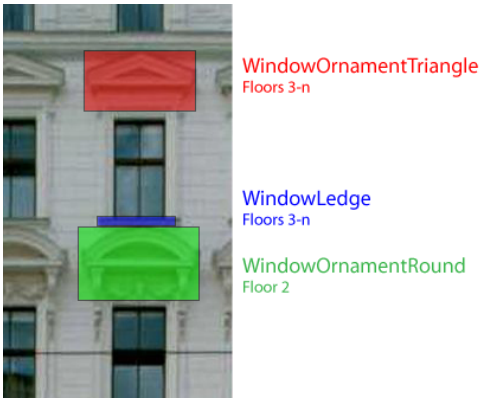
2. Add the following rule to scale, position, and insert your window asset and a glass plane behind:

```
Window -->
        s('1,'1,0.2) t(0,0,-0.2)
        t(0,0,0.02)
        [ i(window_asset) Wall(0) ]
        Glass
```



Window ornaments

Looking at the facade photo again, you notice there are different windows (or window elements) on the different floors.



You need to extend your WindowTile rule and trigger shapes specific to the floor index as follows:

- No special ornaments on the first and the top floors (indices 1 and 999); consequently, only Window is invoked.
- On the second floor, you'll insert an additional Shape WindowOrnamentRound. Because this element is to be aligned to the top border of the window, you translate the current Scope upwards the y-axis with '1.
- The other window tiles (on the mid floors) get an additional WindowLedge shape, as well as the WindowOrnamentTriangle ornament, again translated along y.

```
WindowTile(floorindex) -->
        case floorindex == 1 || floorindex == 999: Window
        case floorindex == 2 : Window t(0,'1,0) WindowOrnamentRound
        else : Window WindowLedge t(0,'1,0) WindowOrnamentTriangle
```

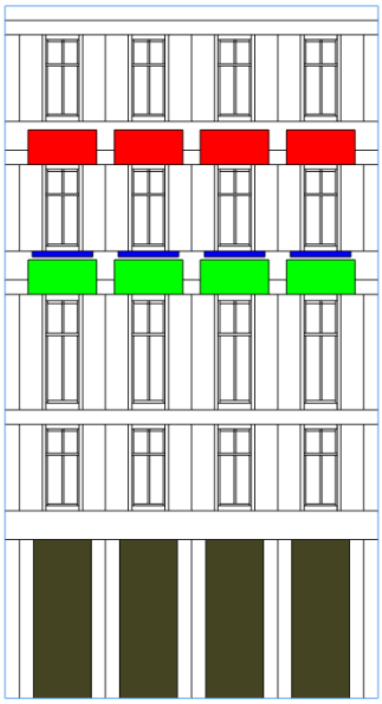
Rather than using the final assets directly, you'll insert proxy cubes first. This makes it easier to set the dimensions for the real assets. You can use the built-in cube asset for this case.

Set the dimensions, center the scope on the x-axis, insert the cube, and color it for better visibility.

```
WindowOrnamentTriangle -->
        s('1.7, 1.2, 0.3) center(x) i("builtin:cube")  color("#ff0000")

# set dimensions for the triangle window element and insert it
WindowOrnamentRound -->
        s('1.7, 1.2, 0.4) center(x) i("builtin:cube") color("#00ff00")

WindowLedge -->
        s('1.5, 0.2, 0.1) t(0,-0.2,0) center(x) i("builtin:cube")
color("#0000ff")
```

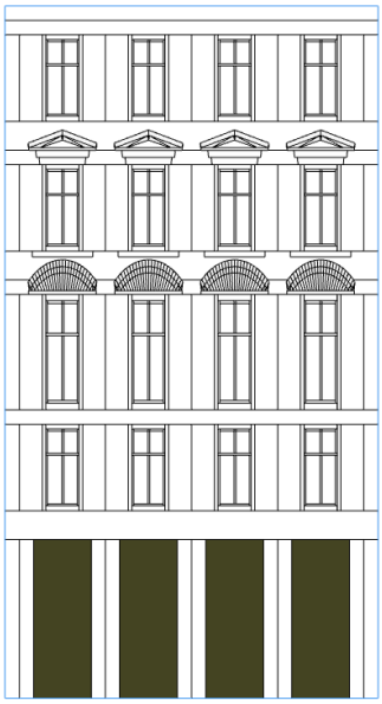


Exchange proxies for real assets

Dimensions of your window ornaments seem reasonable, so you can insert the actual assets instead of the cube (for the WindowLedge, keep with the cube).

```
WindowOrnamentTriangle -->
        s('1.7, 1.2, 0.3) center(x) i(tri_wintop_asset)

WindowOrnamentRound -->
        s('1.7, 1.2, 0.4) center(x) i(round_wintop_asset)
```



You're not finished with the windows on the second floor: The round window ornaments are missing the side pillars. You need to add a new split command to the WindowOrnamentRound rule you previously added. This line will prepare the scopes for the following modillion asset.

```
WindowOrnamentRound -->
               s('1.7, 1.2, 0.4) center(x) i(round_wintop_asset) Wall(0)
               split(x){~1 : WindowMod | window_width : NIL | ~1 : WindowMod }
```

Dimensions are set, and the modillion asset is inserted. Note that by applying the relative negative translation ('-1) in y-direction, the asset's top is aligned to the bottom face of the ornament.

```
WindowMod -->
        s(0.2,'1.3,'0.6) t(0,'-1,0) center(x) i(modillion_asset) Wall(0)
```

Doors

The door tile is split vertically into door, arc, and arctop areas.

To ensure non-elliptic arcs, the height of the arcs area needs to be half the width of the door (the current x scope).

```
DoorTile -->
        split(y){~1 : Door | scope.sx/2 : Arcs | 0.5 : Arctop}
```

On the top area, a wall element and an overlayed modillion asset are inserted.

```
# Adds wall material and a centered modillion
Arctop -->
        Wall(1)
        s(0.5,'1,0.3) center(x) i(modillion_asset) Wall(1)
```

The arcs area is split again, and two arc assets are inserted. Use the wall_inset variable you defined earlier. You need to rotate the right halfarc to orient it correctly.

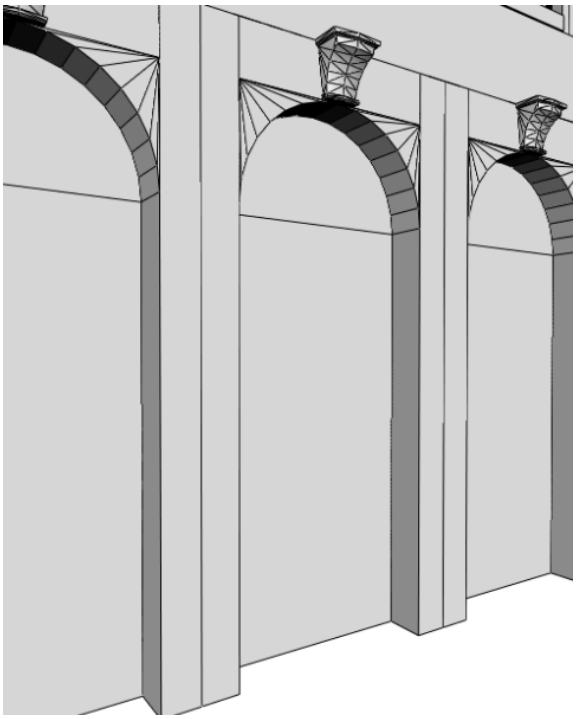
```
Arcs -->
        s('1,'1,wall_inset) t(0,0,-wall_inset)
        Doortop
        i("builtin:cube")
        split(x){ ~1 : ArcAsset
                       | ~1 : r(scopeCenter,0,0,-90) ArcAsset}
```

Set the Wall shape on Doortop and Door, and insert the actual arc asset.

```
Doortop -->        Wall(0)

Door --> t(0,0,-wall_inset) Wall(0)

ArcAsset --> i(halfarc_asset) Wall(1)
```



Ledges

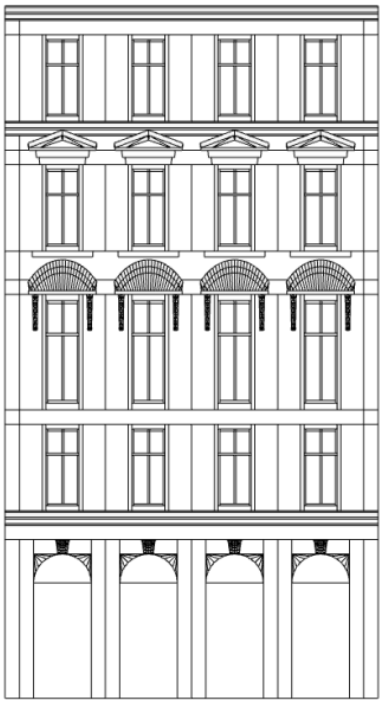
For the ledges, you need rules for top and bottom ledges. Top ledges use a simple wall stripe, bottom ledges need to be distinguished on the different floors with the actual ledge asset inserted.

```
TopLedge --> WallStripe

BottomLedge(floorindex) -->
        case floorindex == 1 : split(y){~1 : Wall(0) | ~1 : s('1,'1,0.2) LedgeAsset}
        case floorindex == 999 : split(y){~1 : WallStripe | ~1 : s('1,'1,0.2) LedgeAsset}
        else : WallStripe

WallStripe --> split(x){ 0.5 : Wall(1) | ~1 : Wall(2) | 0.5 : Wall(1) }

LedgeAsset --> i(ledge_asset) Wall(0)
```



Balcony

Now you'll work on the balcony.

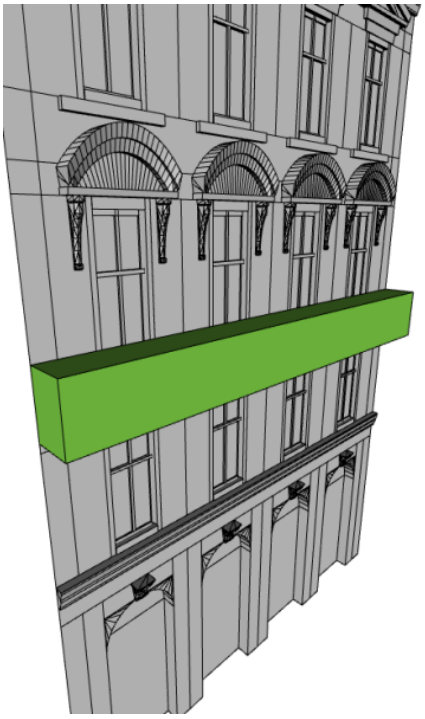
Steps:

1. Use the Floor rule, and add the Balcony shape to the second floor case.

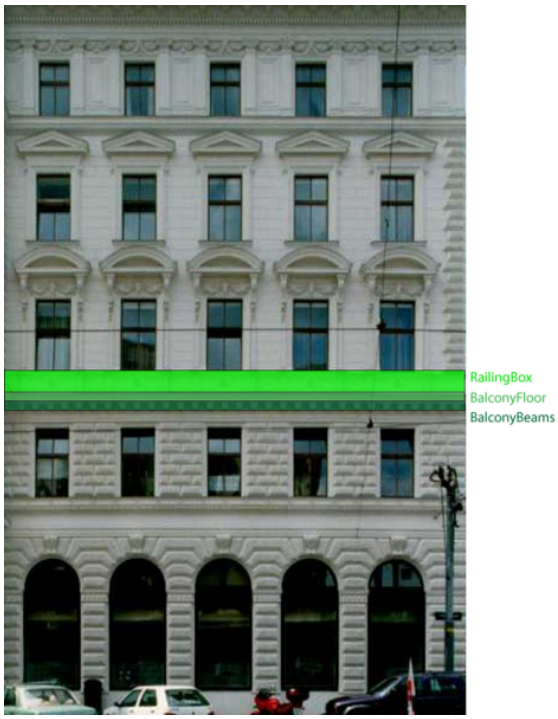
```
case floorindex == 2 :
                split(y){~1 : Subfloor(floorindex) Balcony | 0.5 : TopLedge}
```

2. Start with a simple proxy to ensure the placement and dimensions of the balcony.

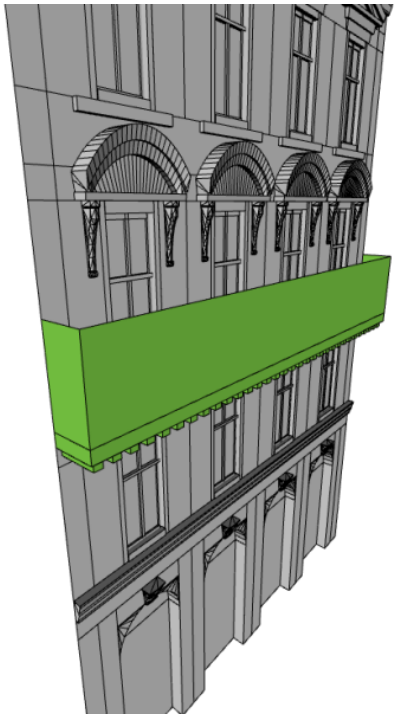
```
Balcony -->
       s('1,2,1) t(0,-0.3,0) i("builtin:cube") color("#99ff55")
```



3. The balcony box is now split into its components: beams, floor, and railing.



```
Balcony -->
       s('1,2,1) t(0,-0.3,0) i("builtin:cube")
       split(y){0.2 : BalconyBeams
                    | 0.3 : BalconyFloor
                    | 1 : RailingBox }
```



4. The beams supporting the balcony are created with a repeating split.

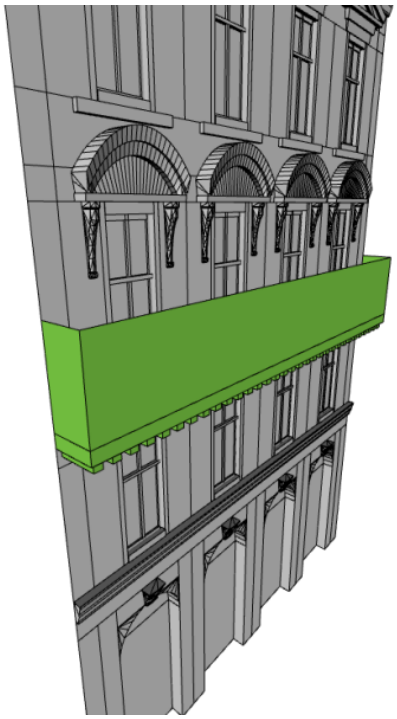
```
BalconyBeams -->
       split(x){ ~0.4 : s(0.2,'1,'0.9) center(x)  Wall(0) }*
```

The BalconyFloor shape only triggers the Wall rule.

```
BalconyFloor --> Wall(0)
```

5. Using the component split on the RailingBox, extract the necessary faces for the balcony railings.

```
# Get the front, left and right components (faces) of the RailingBox shape
RailingBox -->
       comp(f){front : Rail | left : Rail | right : Rail}
```

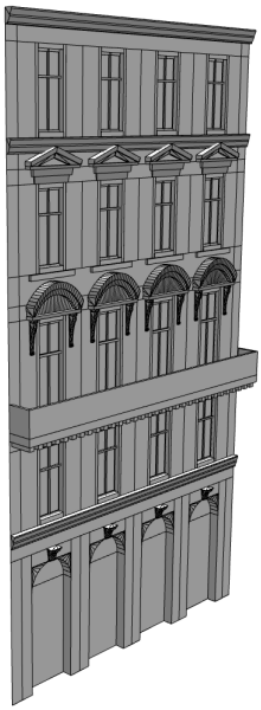


6. Set the dimension insert to a cube to create the balcony rails.

```
Rail -->
        s('1.1,'1,0.1) t(0,0,-0.1) center(x) i("builtin:cube") Wall(0)
```

Final facade with geometry assets

Now you have the final model. Apply the rule to different lot shapes, or explore the user attributes in the Inspector to modify your facade design.



In the next section, you'll learn how to apply textures to this facade.

Texture the facade

This section shows techniques to texture your facade.

Steps:

1. Open the `Tutorial_07_Facade_Modeling/scenes/FacadeModeling_03.cej` scene file if it's not already open.
2. Open the `Tutorial_07_Facade_Modeling/rules/facade_03.cga` file.

Texture assets

Steps:

1. On top of the rule file, add the textures that are going to be used as attributes.

```
const wall_tex                              = "facades/textures/brickwall.jpg"
const wall2_tex                              = "facades/textures/brickwall_bright.jpg"
const dirt_tex                              = "facades/textures/dirtmap.15.tif"
const doortop_tex                       = "facades/textures/doortoptex.jpg"
```

2. For the window and door textures, you'll use a function to get the texture string so you don' need to list the textures separately.

```
randomWindowTex = fileRandom("*facades/textures/window.*.jpg")
randomDoorTex = fileRandom("*facades/textures/doortex.*.jpg")
```

Set up the global UV coordinates

Texturing with the shape grammar consists of the following three commands:

- setupProjection()—Defines the UV coordinate space
- set(material.map—Sets a texture file
- projectUV()—Applies the UV coordinates

You'll add two texture layers to the facade: a brick texture and a dirt map. To maintain consistent texture coordinates over the entire facade, you need to add the UV setup to the Facade rule. To test the texturing setup beforehand, you'll add a new intermediate FrontfacadeTex rule.

Steps:

1. Change the Building rule to the following:

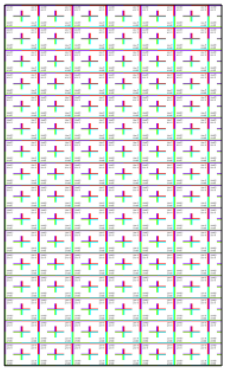
```
Building --> comp(f) { front : FrontfacadeTex}
```

2. Create the following new rule:

```
FrontfacadeTex -->
        setupProjection(0, scope.xy, 2.25, 1.5, 1)
        setupProjection(2, scope.xy, '1, '1)
        Frontfacade
```

   setupProjection(0, scope.xy, 2.25, 1.5, 1) defines the texture coordinates for the texture channel 0 (the color channel). The UV coordinates will be projected along the scope's xy plane and repeated every 2.25 units in x and every 1.5 units in y direction. Set the colormap to builtin:uvtest.png, which is a texture to quickly check UV coordinates. Then apply the UV coordinates by baking the UV coordinates for channel 0.

3. Generate the facade to see the UV setup.



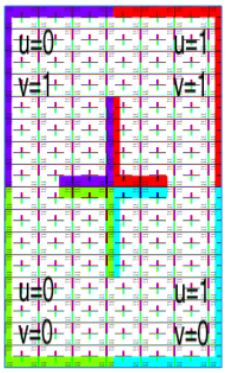
4. Add the UV setup for the dirt channel.

```
FrontfacadeTex -->
        setupProjection(0, scope.xy, 2.25, 1.5, 1)
        texture("builtin:uvtest.png")
        projectUV(0)

        setupProjection(2, scope.xy, '1, '1)
        set(material.dirtmap, "builtin:uvtest.png")
        projectUV(2)
```

   This texture should span the entire facade, so you'll use the relative operators '1 and '1 for the UV setup, which are the dimensions of the facade.

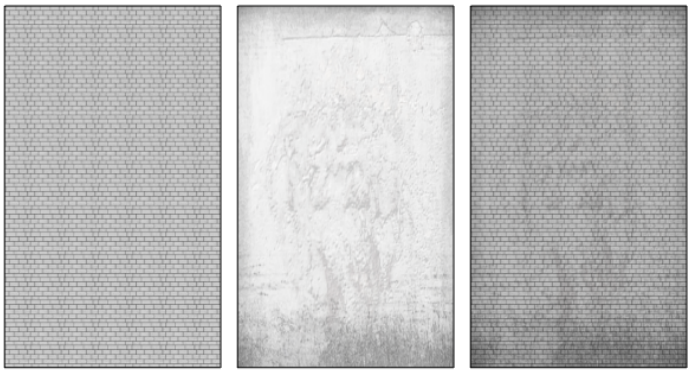
5. Generate the facade again to get the following result:



6. To see how the facade will look with the actual textures, exchange the built-in uvtest texture with the real ones.

```
FrontfacadeTex -->
	setupProjection(0, scope.xy, 2.25, 1.5, 1)
	texture(wall_tex)
	projectUV(0)

	setupUProjection(2, scope.xy, '1, '1)
	set(material.dirtmap, dirt_tex)
	projectUV(2)
```



The UV coordinates are appropriate for the facade.

7. For the actual building, you only need the UV's setup at this point, so change the FrontfacadeTex rule to the following:

```
FrontfacadeTex -->
	setupProjection(0, scope.xy, 2.25, 1.5, 1)
	setupProjection(2, scope.xy, '1, '1)
	Frontfacade
```

The old Frontfacade rule now has UV coordinates set up correctly for the subsequent elements.

Texture the walls

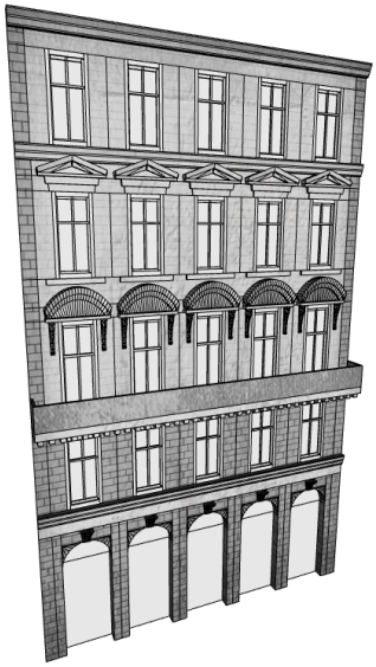
Recall that you added a type parameter to the Wall rule. You'll use that now to get three wall types with different textures.

Steps:

1. Change the Wall rule to the following:

```
Wall(walltype) -->
	// dark bricks with dirt
	case walltype == 1 :
		color(wallColor)
		texture(wall_tex)
		set(material.dirtmap, dirt_tex)
		projectUV(0) projectUV(2)
	// bright bricks with dirt
	case walltype == 2 :
		color(wallColor)
		texture(wall2_tex)
		set(material.dirtmap, dirt_tex)
		projectUV(0) projectUV(2)
	// dirt only
	else :
		color(wallColor)
		set(material.dirtmap, dirt_tex)
		projectUV(2)
```

All wall elements are textured.



Texture the window asset

For the window asset, you'll use a set of window textures to color the glass pane, so you need to set up the UV coordinates to span the entire glass shape. This is done by using '1 for both the x and y directions.

Steps:

1. Call the randomWindowTex function you defined earlier for the texture call.

```
Glass -->
        setupProjection(0,scope.xy, '1, '1)
        projectUV(0)
        texture(randomWindowTex)
```

2. Add specular luster to the glass.

```
Glass -->
        setupProjection(0,scope.xy, '1, '1)
        projectUV(0)
        texture(randomWindowTex)
        set(material.specular.r, 1) set(material.specular.g, 1)
      set(material.specular.b, 1)
        set(material.shininess, 4)
```

Texture the door shapes

The door planes are textured in the same way as the window glass.

```
Doortop -->
        setupProjection(0, scope.xy, '1, '1)
        texture(doortop_tex)
        projectUV(0)
Door -->
        t(0,0,-wall_inset)
        setupProjection(0,scope.xy, '1, '1)
        texture(randomDoorTex)
        projectUV(0)
```