# Cga-LoD – Developer Documentation

## Introduction

Cga-LoD was programmed 2009/10 as a Praktikum. The purpose of the program is to create LoD models of models that were created with Markus Lipp's Cga program.

Markus Lipp's is used to create models of buildings, and in doing so hierarchical rules that describe the building and can automatically create the model. A (simplified) version of this rule-hierarchy can be seen in figure 1.

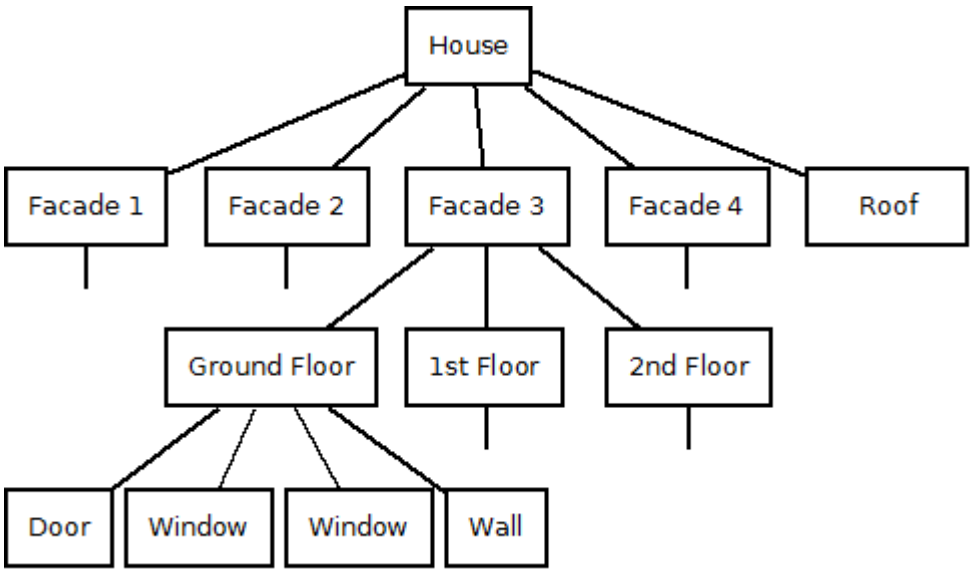


*Figure 1: An example of the hierarchical rules of the Cga program.*

The basic idea of Cga-LoD is to use this hierarchy and create the LoD model by replacing parts of this hierarchy (whole sub-trees to be precise) by oriented bounding boxes that enclose the geometry of this parts. Later on this oriented bounding boxes will be texturized by projecting the geometry onto the box faces.

## Basic Structure of the program

The program is basically structured into three components:

- CgaLoDGui
- CgaLoD
- Luxor

CgaLoDGui was supposed to contain all code that belongs to the creating a window and the user interface, while CgaLoD was supposed to contain all the code needed to create and render a LoD version of a Cga model. With this separation it should have been easier so change the user interface if needed (but since it wasn't implemented perfectly there still would have been some work to do).

The Luxor component is basically just a general collection of tools that can be used for rendering.

The source code for CgaLodGui can be found in the 'Application' subfolder of the source folder, CgaLoD lies in 'CGA_LOD', and Luxor lies spread over all other subfolders.

Entrypoint of the program lies in the main.cpp file, and actually just calls the Run() method of the ApplicationManager.

## CgaLoD

CgaLoD contains a small scene, basically build out of a Camera, a Light and the LoDMesh. The scene (class Scene) is used to access all components of this components, mainly the the LoDMesh.

The LoDMesh contains a LoD version of a Cga model, and is therefore the most important class of the program. LoDMesh allows to load a Cga model, create a LoD version out of it, render the LoD/original version and save the LoD version into a fbx file.

Another important class is the TexturePartitioner. This class is used by the LoDMesh to create texture coordinates and textures, that will be used to texturize the bounding boxes of the LoD version. The texture partitioner stores several textures, and splits them into small rectangular areas when the LoDMesh texturizes the bounding boxes. This way just a small number of textures are needed, though there might be a lot of bounding boxes.

### Loading Cga models

The Cga program originally contained one method to save a created model into an fbx file. But thie method grouped the geometry of this model by textures and thereby destroying the hierarchy. Therefore a second method was implemented, which does not group the geometry and keeps each geometry in a separate node when exporting as an fbx file.

To identify each node in the fbx file, we use the nodes name to store the information about its position in the hierarchy. This information could look like this:

```
I/0 wallA/2 Subdiv/0 groundFloorTileA/0 Subdiv/1 cndGroundFloor/0
Subdiv/3 candlerFacade/0 Comp/1 houseType/0 house
```

As a delimiter blank is used, the last token at the end (in this example "house") is the name of the root of the hierarchy.

Additionally to the fbx file, an xml file is exported (filename just extended by ".xml", so "A.fbx" will have an "A.fbx.xml" file). In this xml file we can find information about the oriented bounding boxes (which are known to the Cga program, but wouldn't be known to the Cga-LoD program) of each node.

The xml file will contain information for all nodes in the hierarchy, while the fbx file will just contain those nodes that have geometry stored in them.

### Selecting Bounding Boxes

To create the LoD model we have to partition the nodes of the hierarchy into tree groups: nodes whose original geometry we use, nodes whose bounding box we use and nodes we don't use at all (since they are covered by another node whose bounding box is used).

To do this we assign each node a level value and compare each nodes level value to a level threshold (called Current Level), see figure 2. When a nodes level value is above or equal to the threshold, we use the original geometry. But if it is below the threshold we ignore the node or use its bounding box (if it is the root of a sub-hierarchy).

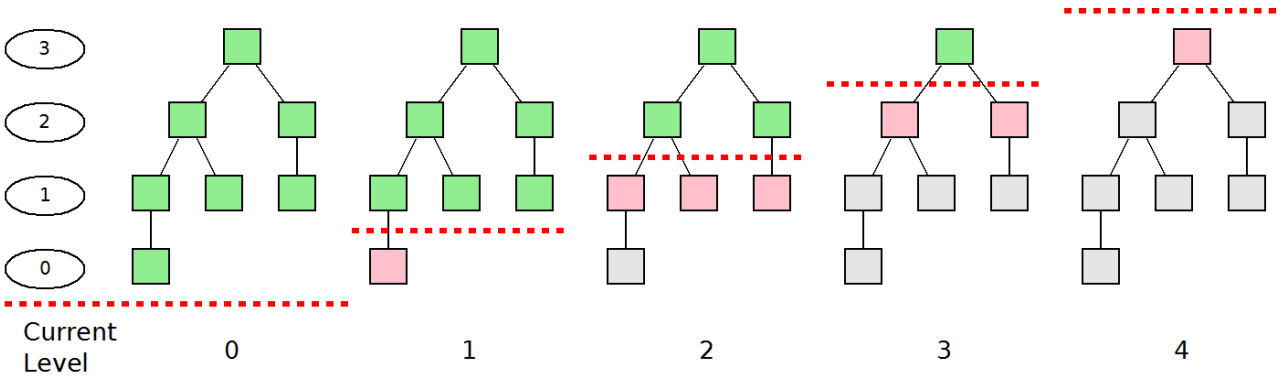


*Figure 2: The bounding boxes are selected by comparing the level value of each node in the hierarchy to the current level. Green nodes use the original geometry, red nodes use bounding boxes and grey nodes are not used at all (but for each grey node there is a red node)*

To calculate the level values, we have basically two sets of methods, the ones based only on the hierarchy and the ones based on the OBBs.

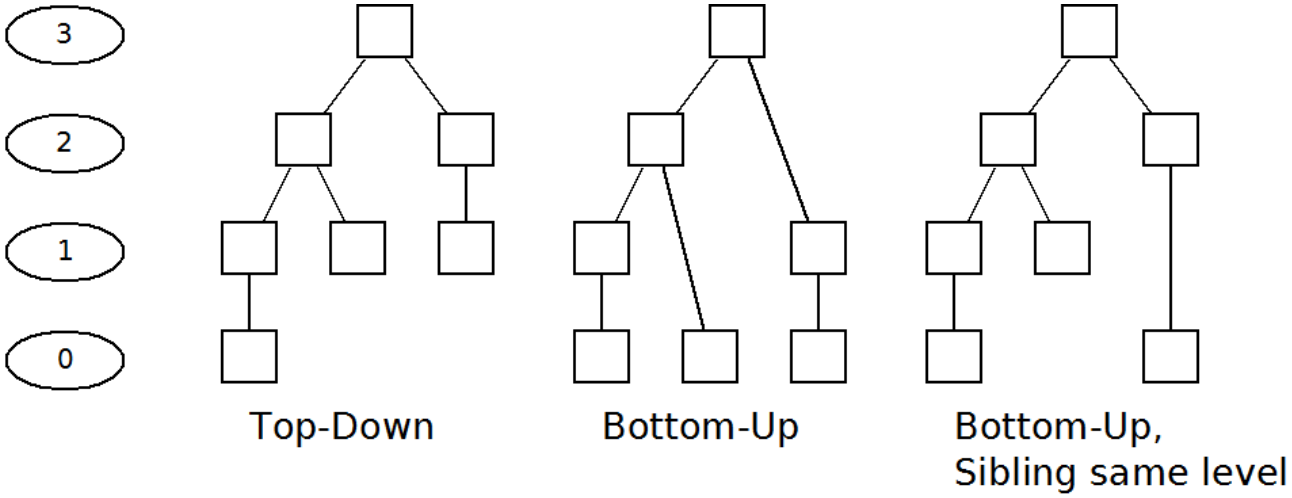


*Figure 3: Calculation of the level value based on hierarchy only.*

As can be seen in figure 3, when using hierarchy only we can use a top-down approach by assigning the root node the maximum level value, and decrease it by one for each child. Or we can use bottom-up, and give all nodes a level value as small as possible (leafs have level 0). Or of course we can modify bottom-up by assuring that sibling nodes have the same level value.

The second set of methods to calculate the level value is based on the bounding boxes. For each node we calculate a value v (e.g. the length of the diagonal of the bounding box), and then use this value to calculate a level value for each of the nodes (by interpolating between the minimum and maximum v). There are several methods to calculate v, basically all rely on the dimensions of the bounding box (x, y and z) and a function (minimum, maximum, sum, euclidean metric).


*Figure 4: Calculation of the level value based on the bounding boxes.*

## Creating the LoD version

The creation of the LoD version happens in the update method of LoDMesh. When the LoDMesh receives new parameter values (by its setter methods) it sets flags that the LoDMesh to create a new LoD version the next time update is called. Therefore it has to be said that there are always two versions of the parameter values. The ones that are displayed in the UI and the ones that are currently stored in the LoDMesh.

The update flags are not binary (update or no update), but tell the update what it has to to. There are a total of five flags, each referring to one of five more update methods. When a flag is set, its update is called, otherwise not. Since not all parameters affect all five update methods, this way not all of them have to be called. The current set flags are stored in m_whatsToUpdate and can be changed with the enum UpdateMode.

The five update functions are:

- UpdateLevelMap()
- UpdateNodeCache()
- UpdateTexCoords()
- UpdateGeometryCache()
- UpdateTextures()

They have to be kept in that order, since the later rely on the results of the previous called updates.

UpdateLevelMap() is used to create new level values. The program offers different methods to create the LoD version (the socalled Level Mode). Each of this methods calculates a different level value for each node of the original mesh. Since calculating this values for a single method is as much expansive as calculating them for all methods, we store all level values in an array, the level

map.

For further methods we need to process all nodes of the hierarchy whose bounding box or whose original geometry is used. Therefore UpdateNodeCache() collects all this nodes in two lists, so further nodes can use them directly and don't have to access them via the hierarchy.

UpdateTexCoords() creates new texture coordinates for all bounding boxes currently used. This method uses the TexturePartitioner.

UpdateGeometryCache() is used to collect and group all vertex geometry (of the original mesh and the bounding boxes) and group them by their textures. Then it merges all this groups into single geometries and stores them in VBOs. This way a lot of geometry can be reduced to a small number of VBOs.

At last UpdateTextures() creates the textures for all bounding boxes. It uses the texture coordinates created by UpdateTexCoords() and the bounding boxes and renders the parts of the Cga model that lie inside the bounding box into the textures (to be precise, the parts assigned to the bounding box by the TexturePartitioner).

## Texture partitioning

As mentioned before we use the TexturePartitioner to split the LoD textures into small parts that are used to texturize the sides of the bounding boxes. When using the TexturePartioner we don't create partition for all box sides at once, but for one after another. This doesn't give a perfect result, but works quiet fine.

The TexturePartitioner basically works on the idea to virtually split each texture into strips, and to cut off end pieces of this strips when a new box side is texturized. This end pieces are then reserved for the box side. Figure 5 shows the basic method, although we don't always need to create new strips/textures, often we can use existing strips/texture.

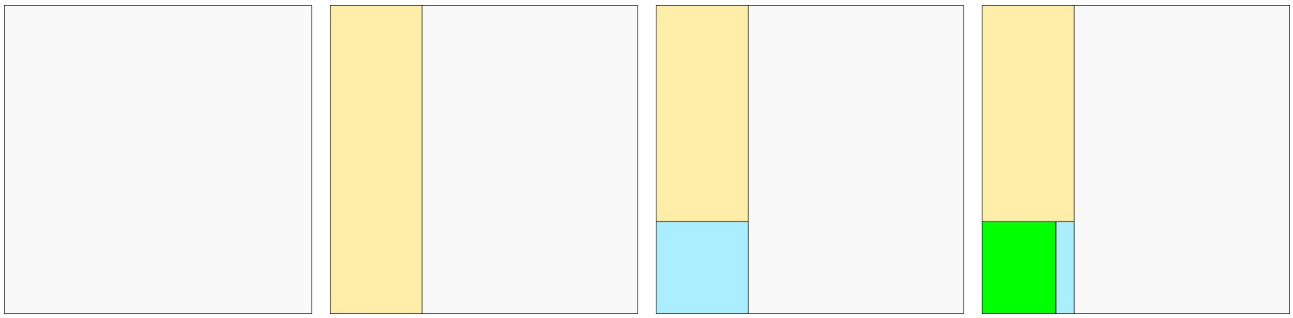

*Figure 5: Creating a new partition. A strip (yellow) is created in an empty texture (white), and a small part of the strip (blue) is virtualy cut off. This part is reserved for the a bounding box side (green), although it might be slightly larger than it would have to be.*

To make the results of the partitioning better, Cga-LoD uses two extensions. First, since box sides are rectangular, we can rotate (in the texture space) by 90°, and therefore might find a better place for the box side to fit (although then, when it comes to filling the textures with colors, this has to be taken into account).

The second extension is to sort all box sides by their largest side descending. When we now partition the textures, the wider strips are created first, and it won't happen that we have to create a large strip later on (which would lead to a worse partitioning).