Connected Meshes

Johannes Unterguggenberger* Student

Martin Ilčík Supervisor

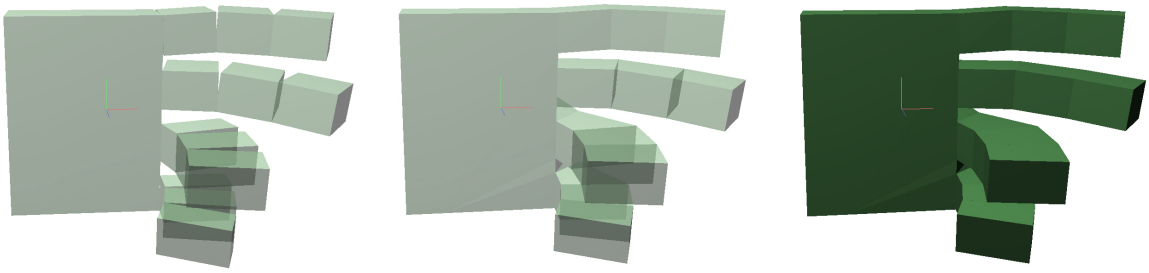

Figure 1: These figures show a shape connected with a shape grammar. The left figure shows the not connected shape. The middle and right figures show the same shape with connected vertices. The geometry still consists of separate polyhedrons, but their vertices have been changed to a connection-point's coordinates. The separate polyhedrons can be seen at the middle figure.

Abstract

This report presents techniques to connect 2-dimensional and 3-dimensional shapes. These techniques have been developed to connect basic shapes which are generated by a shape grammar for computer generated architecture, in particular for CGA shape. Consider an arm: the shape grammar creates two cuboids - one for the upper arm and one for the lower arm. When the arm is bent, the two cuboids should be connected at the elbow. How to do that, where to store the connectivity information, and different connection-strategies are the main topics of this report.

Keywords: shape grammars, procedural modeling, shape connectivity

1 Introduction

Humanoid characters are widely used in today's computer graphic applications like games, movies, or simulations. The classical approach to model humanoid characters is to use a 3D-modelling tool and create the geometry manually. If the model should be animated, a skeleton has to be added to the mesh. The model's geometry is altered based on the skeleton (bone rotations, etc.). [Badler et al. 1993], [Ratner 2003]

Manually modelling a humanoid model or creating variations of an existing one can be very time-consuming. A procedural approach to create humanoid models based on a shape grammar called CGA shape is presented in [**?**] and [Fiedler 2009]. CGA shape [Müller et al. 2006] has been developed specifically for the automatic design of computer-generated architecture. With a grammar-based procedure, creating variations of existing models is just adjusting a few parameters, and yet the resulting humanoid model's stature or body weight has changed, for example. A skeletal system for posing and interactive manipulation of generated models has been added to GCA shape [**?**].

So far, the modified CGA shape grammar produced models consisting of shapes which are connected with a kinematic skeleton. However, these shapes are not connected geometrically (see figure 1). How to connect these shapes is addressed by this report. In section 2 we discuss where to store the connection information. Section 3 presents the geometry adaption concept to reconnect meshes after moved apart. After these two preparing steps, in section 4 the algorithms for connecting 2D and 3D geometry are explained in detail. This involves some math [Papula 2007], which is explained in section 5.

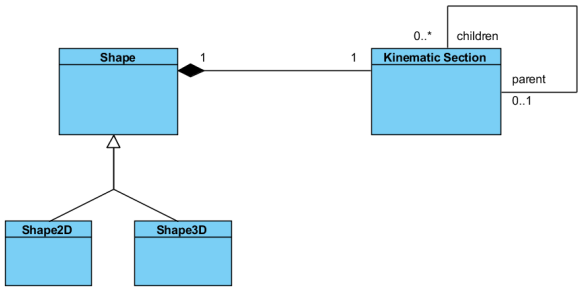

Figure 2: Class diagram showing the CGA shape, and kinematic section

2 Store the connection information

Figure 2 shows the parts of the CGA framework which are relevant for the decision where to store connectivity information. A shape always has an associated kinematic section. The kinematic section contains parent and child relations. A shape is aggregated with one kinematic section, there is a 1:1 relation between them. Because of that and because the connectivity information needs to reference the shape's geometry, we have decided to store the connectivity

*e-mail: johannes.unterguggenberger@gmail.com

information directly with the shape.

Connectivity information means a mapping from a parent shape's vertex to a vertex of one of the children's vertices, which it is connected to. Such a parent-to-child mapping is stored in a child shape. This is the most convenient place to store the mapping. If a parent has more child shapes, which could be the case after a parallel split, separate connectivity information has to be stored for each child. By storing the vertex mapping in the child shape, no further data structures are needed. The tree representing parent-to-child relations is already there in form of the associated kinematic section and its parent and child references.

Connectivity preservation

The vertex mappings are weak references - the vertex indices of the shape's geometry are stored in the vertex mapping. Therefore, it has to be taken care of updating that mapping whenever the geometry changes so that the vertex indices change. This could happen, when a subsequent sequential split is applied to a shape - the vertex indices will most probably be different after the split.

In order to preserve the connectivity, whenever a split is applied to a shape the vertex mappings have to be updated to the new vertex indices. Also all of that shape's children have to be updated to the new vertex indices. The parent shape's vertex mappings do not have to be updated because connectivity information is stored in the children. That means the splitted shape contains the vertex mappings from its parent to itself and only its own mapping has to be updated. However, if the shape has children, all of their vertex mappings have to be updated as well. It has to be ensured that the mappings are updated whenever an operation has been applied which changes the vertex indices.

See figure 3 for the extended class diagram including the vertex mapping and the weak references to polygon vertex indices in case of a 2D shape, or to polyhedron vertex indices in case of a 3D shape, respectively.

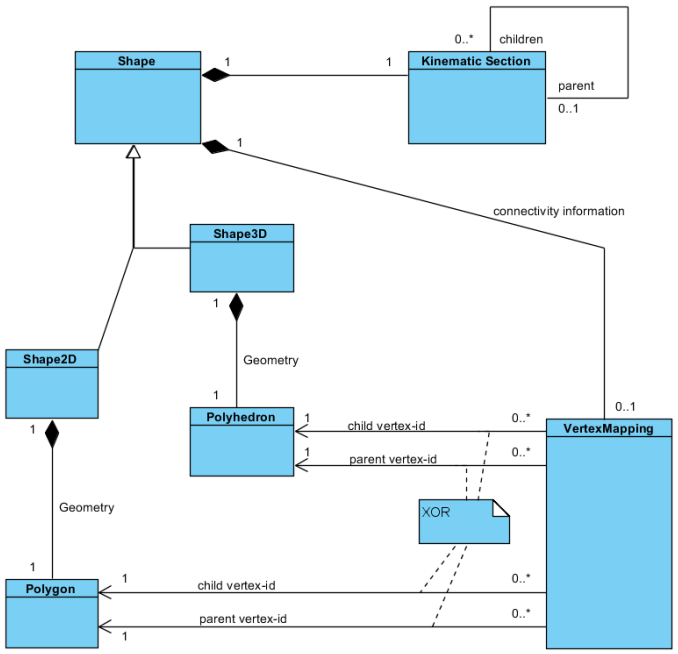

Figure 3: Class diagram including the vertex mapping and the weak references to polygon and to polyhedron

3 Geometry adaption

Applying a split rule on a shape could create geometry similar to that shown in figure 4. No rotations or scales are applied on the middle shape, which has the effect that both, the parent shape's vertices and it's child shape's vertices have exactly the same coordinates. The right shape has a rotation applied. How to connect the associated vertices is described in section 4. Before the connection algorithm is applied, it has to be ensured that both shapes to be connected have the same amount of vertices on the connection-facet. After a sequential split, this is the case innately since the sequential split does not change geometry but split the existing geometry.

It is more complicated with a parallel split. After a parallel split has been applied on a child shape, usually new parent geometry has to be created. See figure 5. After the parallel split the parent shape has 4 child shapes. In order to connect each of the 4 child shapes to the parent shape, the parent's geometry must have as many vertices along the connect-line as all of the children together along the connect-line.

The new parent geometry is generated right after the parallel split is applied and before any rotations or other transformations are applied. At this time the positions of all child shapes are known and additional parent vertices are created at the positions of the corresponding vertices of the child shapes. The mappings are stored in the child shapes and used during rendering for doing the connections.

Creating the new vertices is quite simple. The algorithm moves along the parent's poly-line in counter clockwise direction until the connect-line is found, which is then replaced by several shorter lines. The vertex positions for these shorter lines come from the child shapes. Start-point and end-point have exactly the same coordinates as the two connected vertices in the child shape. To get them in the right order, the child's poly-line is traversed in clockwise direction until both connected vertices have been found.

Geometry creation in 3D

Creating new geometry for 3D-shapes is even more challenging because it is not just adding a few additional points to a polyline. In the 3D case, actually facets are connected to each other (despite still their vertices are stored as connection-information). Therefore, new facets have to be added to the geometry. Figure 6 shows a way of creating new, valid connection-geometry, which is implemented by our implementation. In that example, the facet facing in the opposing direction remains unchanged. This is good if the shape is connected to a parent shape. Generally, all facets remain the same except for the connect-facet and the two side-facets of the connect-facet.

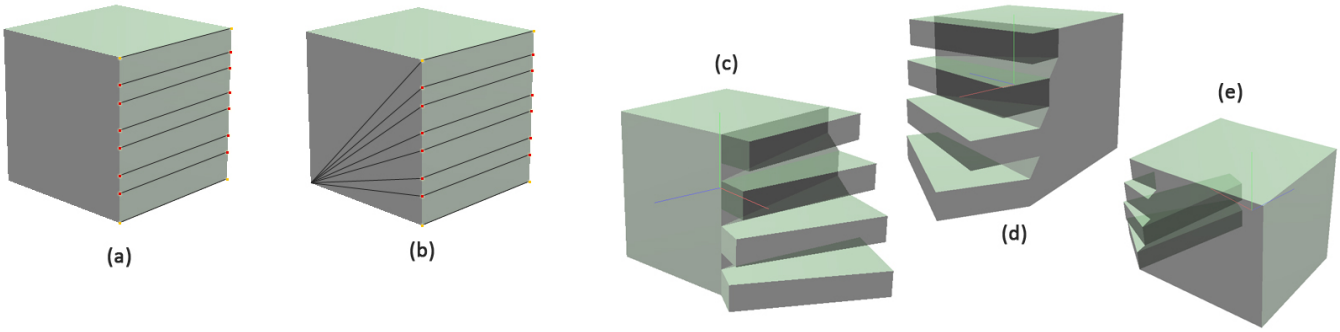

Figure 6: In order to add connect-facets to the parent geometry, vertices have to be added along the edges on both sides of the original facet in split direction - the red vertices in figure (a). The orange vertices are also needed, but they are already there in the original polyhedron. Between those newly created vertices facets are created. This, however, does not yield a valid closed polyhedron. The side facets have to be replaced by new ones, too, as illustrated in figure (b). The geometry of the back-facing facet is not changed to avoid any problems with eventually existing connectivity to a parent-shape. Figures (c), (d), and (e) show an example of a connected parallel split with the parent shape from figure (b) from different perspectives.

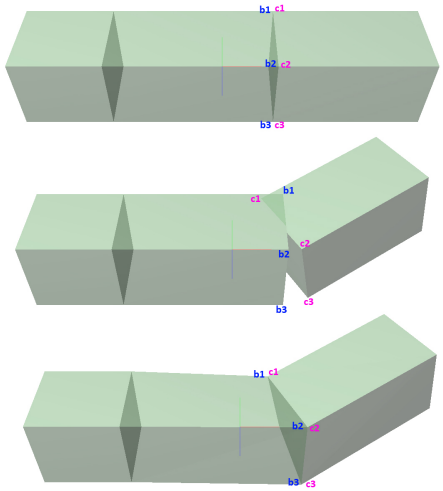

Figure 4: The topmost image shows an already splitted shape without any rotations or scalings applied to the parts. The vertices numbered with b1, b2, b3 should be connected to c1, c2, c3 in this order. The middle image shows the result with the last part rotated but not connected. The bottom image shows the shapes with the same rotation and with their vertices connected.

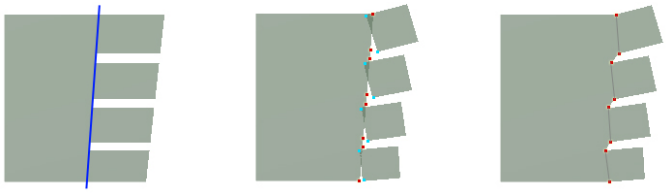

Figure 5: The left image shows a parent shape which has 4 children. These were created via parallel split. Immediately after the split all vertices that should be connected lie on the blue line. In the middle image rotations are applied to all of the child shapes. In order to connect them to the parent, additional vertices have to be added to the parent along the connect-line. The red dots represent the positions where the vertices have to be added - exactly at the positions where the cyan vertices of the child shapes were before the rotation. The right image shows the final result with connected shapes. The parent's (red) vertices and the corresponding children's (cyan) vertices have been moved to the same positions.

4 Connecting the vertices

In the previous sections preparing the shapes for connecting has been discussed. In section 2 the right place to store the connection-information (also referred to as vertex-mapping) has been investigated. The essence of the previous section 3 is that for some cases the shape's geometry has to be extended in order to get enough vertices to connect all child-shapes to.

In this section the actual connecting-algorithms are explained. Connecting two shapes means moving their mapped vertices both to the same new position. This position is the connect-point. In order to find that position, several different methods will be introduced, which are called *shape-connectors*. The shape-connectors for 2D shapes and for 3D shapes work differently, of course. The same types of shape-connectors exist for 2D and 3D, which there are:

- child-to-parent-connector
- parent-to-child-connector
- midpoint-connector
- intersection-point-connector

The individual connectors are described under the following sections for the 2D case and the 3D case. What is very important in either case is to follow these steps for rendering both, the parent's and the child's vertex which are to be connected:

1. Convert child vertex to parent's coordinate system
2. Calculate connection point via a shape-connector and update vertex position
3. Convert child vertex back into its coordinate system
4. Render the shape with the updated vertex positions

Connecting a 2D shape

Three connector-types are quite easy to implement. The *parent-to-child-connector* simply moves the parent vertex to the child vertex position.
The *child-to-parent-connector* moves the child vertex to the parent vertex position.
And the *midpoint-connector* calculates the point which is in the middle of the parent vertex and the child vertex using the formula:

$$\vec{m} = \begin{pmatrix} \frac{p_x + c_x}{2} \\ \frac{p_y + c_y}{2} \end{pmatrix} \tag{1}$$

where $\vec{m}$ is the midpoint, p is the parent vertex, and c is the child vertex, each of the vertices is represented by a 2-dimensional vector.

The *intersection-point-connector* searches for the only line adjacent to the vertex whose second endpoint is not in the vertex-mapping and creates the line equation $y = kx + d$ based on the two endpoints (one of which is the vertex to be connected), where k is the slope, and d is the y-intersect. Such a line equation is searched for both, the parent shape ($y = k_p x + d_p$) and the child shape ($y = k_c x + d_c$). Finally, the intersection point $\vec{p}$ is calculated:

$$\begin{aligned} p_x &= \frac{d_p - d_c}{k_c - k_p} \\ p_y &= k_p p_x + k_p \\ \vec{p} &= \begin{pmatrix} p_x \\ p_y \end{pmatrix} \end{aligned} \tag{2}$$

Our algorithm uses the *midpoint-connector* if no intersection point could be found as a fall-back. This can happen if the lines are parallel. See figure 7 for more details on the 2D *intersection-point-connector*. If no line can be found with one endpoint in the mapping and the other endpoint not in the mapping, an approximation or rather an assumption is made: The line orthogonal to the line between the current vertex and the following connected vertex is used to intersect with the other shape's selected intersection-line. Figure 8 shows the different results depending on the connector applied.

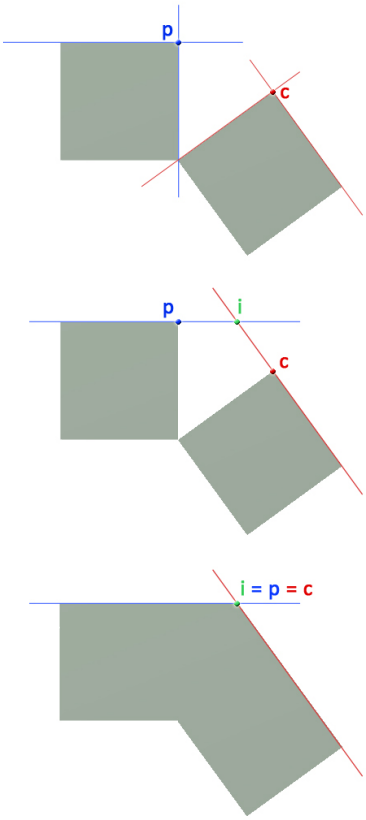

Figure 7: The intersection-point-connector first searches the right edges. Once found, the parent-edge and the child-edge are extended to find their intersection point. The top image shows all adjacent lines to the vertices to connect, which there are the parent's vertex p and the child's vertex c. Our algorithm chooses the adjacent line with one endpoint in the vertex-mapping and the other point not in the mapping. The other candidate has both endpoints in the mapping. This algorithm works for all sequentially splitted shapes. The middle figure shows the lines chosen and their intersection point. The positions of the parent vertex p and the child vertex c are moved to that intersection point's position. The bottom figure shows the final result.

Connecting a 3D shape

In the 3D case, the *intersection-point-connector* is quite involved. The other three connector types are rather easy to implement in 3D, too. The *parent-to-child-connector*, and the *child-to-parent-connector* simply move the vertex to the parent's or child's vertex position, respectively. The *midpoint-connector* simply averages the x, y, and z components of the two vertices. The different results of the several connectors applied are shown in figure 8.

The 3D *intersection-point-connector* basically works similar to the 2D counterpart. The big difference to 2D is that the two lines found (the extended edges of both shapes) can not be intersected with each other to find the intersection point, because in the general case they won't intersect in the 3-dimensional space. To find the edge to be extended, we have to find two facets adjacent to the vertex, which have 2 vertices not in the connection-mapping and the third is the vertex to connect. From these two facets, plane equations are generated. Those 2 planes are then intersected, and their intersection line is the extended edge. Each vertex has at least 3 adjacent facets. In case of an sequential split and well formed shapes, exactly one of those facets is the connect-facet (the facet which should be connected to the other shape), i.e. there are 2 more facets which we can find the plane equation for. Doing this for both, the parent and the child shape, yields the extend-edge for the parent-shape and the extend-edge for the client-shape. See figure 9 to get a better understanding.

The child's line is intersected with up to 2 planes from the parent shape and, vice versa, the parent's line is intersected with up to 2 planes from the child shape. This yields up to

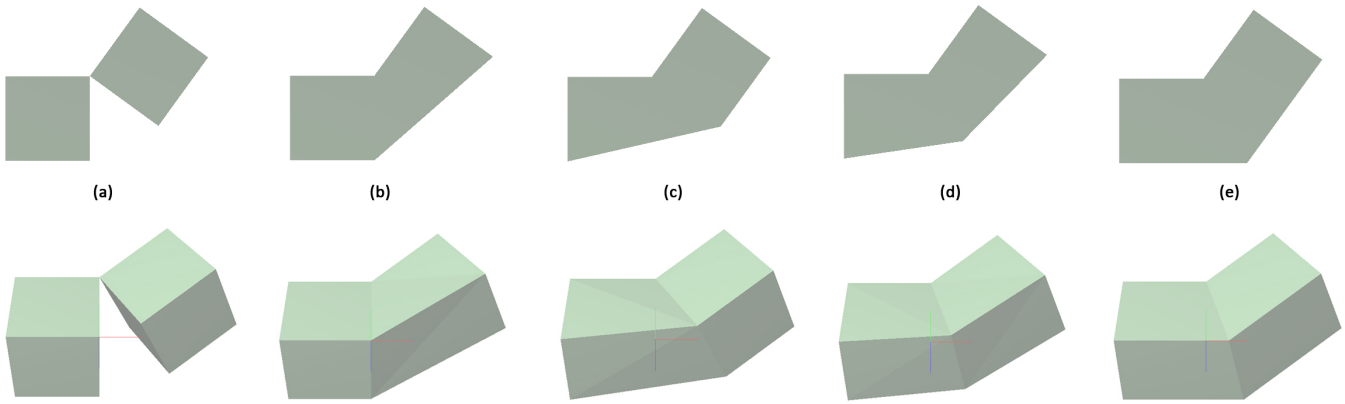

Figure 8: Figures (b) to (e) show the results of the different shape-connectors applied. (a) is the original (sequentially splitted) shape without vertices connected. The top row shows a 2D shape while the bottom row shows a 3D shape. Figure (b) shows the child-to-parent-connector, while in figure (c) the parent vertices are moved to the child's vertex positions using the parent-to-child-connector. (d) shows the midpoint-connector applied to the shapes, and figure (e) shows the intersection-point-connector in action, which basically extends adjacent edges.

4 intersection points, all of which are possible candidates for the intersection point. 2 of those 4 points are eliminated like follows: Of one pair of intersection points, only the intersection point whose distance to the connection-vertices is lower than the other point's distance is chosen. This leaves 2 candidates left (see figure 9).

There are 3 strategies how to get the ultimate connection-point:

- Chose the one of these points whose summed distances to the original parent and child vertices is lower than the other point's distance to them. We call this the *shortest-distance-point*.
- Calculate the *midpoint* of the two intersection-points by simply averaging the x, y, and z coordinates of the two candidates.
- Assume the 2 extended edges to be skew lines, search for the position of the shortest distance between those two lines - their adjoining line. Then take the point in the middle of the adjoining line as the connection-point. We call this the *skew-lines-point*.

It can't be said, that one of these methods is the best for all cases. Depending on how the shapes to be connected are located in the 3D space (rotated, translated, scaled, etc.) to each other, one of those methods will produce the best result, but not always the same method. It depends on the case. Usually, if the constellation is "nicely behaved", the skew-lines-point should be a good choice. But sometimes, the shortest distance between the skew lines is very far away from the vertices to be connected. In that case, our algorithm uses the midpoint. Also when the vertices are very close to each other, the midpoint produces better results. Our current implementation uses a decision tree which chooses the skew-lines-point if the skew lines are not too far away from each other and the skew-lines-point is not much farther away from the vertices than the midpoint. In most other cases, the midpoint is used.

Also for the 3D case exists an approximation to the extended edge. If less than 2 appropriate facets exist for a specific vertex, the facet-normal of the facet to be connected at the position of the vertex to be connected is used instead of the intersection-line of the two planes as an approximation to the extended edge. This usually happens with shapes a parallel split has been applied to.

5 Mathematical background

The mathematical calculations for the 2D shapes are quite easy. Basically, not much more than the line equation $y = kx + d$ is needed. 3D calculations are a little bit more involved, therefore the mathematical utilities used by our implementation are explained here. See also [Papula 2007].

For all calculations, the vectorial representations of lines and planes are used. All vertices on the line $\vec{g}$ are represented by

$$\vec{g} = \vec{o} + \lambda\vec{r} \tag{3}$$

where $\vec{o}$ is the position vector, $\vec{r}$ is the direction vector, and λ is a scalar. All vertices on a plane $\vec{e}$ are represented by

$$\vec{e} = \vec{a} + \beta\vec{b} + \gamma\vec{c} \tag{4}$$

where $\vec{a}$ is the position vector, $\vec{b}$ and $\vec{c}$ are the direction vectors, and β and γ are scalar values. The parameters of a plane in vectorial representation can be created from 3 points of a given facet ($\vec{p_1}$, $\vec{p_2}$, and $\vec{p_3}$) like follows:

$$\begin{aligned} \vec{a} &= \vec{p_1} \\ \vec{b} &= \vec{p_2} - \vec{p_1} \\ \vec{c} &= \vec{p_3} - \vec{p_1} \end{aligned} \tag{5}$$

The normal $\vec{n}$ of a plane is calculated by

$$\vec{n} = \vec{c} \times \vec{b} \tag{6}$$

Now let's see how to compute the intersecting line between two planes with normal vectors $\vec{n_1}$ (plane 1) and $\vec{n_2}$ (plane 2). The line's direction vector $\vec{r}$ is the cross product of both plane-normals:

$$\vec{r} = \vec{n_1} \times \vec{n_2} \tag{7}$$

To get the position vector $\vec{o}$, the following system of equations has to be solved:

$$\begin{aligned} \vec{n_1} \cdot (\vec{o} - \vec{a_1}) = 0 \\ \vec{n_2} \cdot (\vec{o} - \vec{a_2}) = 0 \end{aligned} \tag{8}$$

or with the dot product applied, the equations become:

$$\begin{aligned} n_{1_x}(o_x - a_{1_x}) + n_{1_y}(o_y - a_{1_y}) + n_{1_z}(o_z - a_{1_z}) = 0 \\ n_{2_x}(o_x - a_{2_x}) + n_{2_y}(o_y - a_{2_y}) + n_{2_z}(o_z - a_{2_z}) = 0 \end{aligned} \tag{9}$$

where $\vec{n_1}$ and $\vec{n_2}$ are the normals, and $\vec{a_1}$ and $\vec{a_2}$ are the position vectors of the planes.

A plane and a line have an intersection point, if the dot product of the plane's normal vector $\vec{n}$ with the line's direction vector $\vec{r}$ is different from zero - i.e. $\vec{n} \cdot \vec{r} \neq 0$. Then their intersection point $\vec{g_i}$ can be calculated by:

$$\begin{aligned} \lambda &= \frac{\vec{n} \cdot (\vec{a} - \vec{o})}{\vec{n} \cdot \vec{r}} \\ \vec{g_i} &= \vec{o} + \lambda \vec{r} \end{aligned} \tag{10}$$

where $\vec{a}$ is the plane's position vector, and $\vec{o}$ is the line's position vector. First, λ is calculated which is then used with the vectorial line representation to calculate the intersection point $\vec{g_i}$ with the plane.

Finally, let's see how to find the shortest distance between two skew lines $\vec{o_1} + \lambda\vec{r_1}$, and $\vec{o_2} + \lambda\vec{r_2}$. First, we get the direction $\vec{d}$ of their adjoining line segments by calculating the normalised cross-product of their directions:

$$\begin{aligned} \vec{d} &= \vec{r_1} \times \vec{r_2} \\ \vec{d} &= \frac{\vec{d}}{|\vec{d}|} \end{aligned} \tag{11}$$

Then line 1 is extruded along direction $\vec{d}$, creating plane 1, and line 2 is also extruded along direction $\vec{d}$ which yields plane 2.

Plane 1:

$$\vec{o_1} + \beta(\vec{r_1} - \vec{o_1}) + \gamma(\vec{d} - \vec{o_1}) \tag{12}$$

Plane 2:

$$\vec{o_2} + \beta(\vec{r_2} - \vec{o_2}) + \gamma(\vec{d} - \vec{o_2}) \tag{13}$$

The two endpoints $\vec{p_1}$ and $\vec{p_2}$ of the adjoining line segment can be found by intersecting line 1 with plane 2 and the second endpoint by intersecting line 2 with plane 1.

$\vec{p_1}$ = `intersect(Line1, Plane2)`
$\vec{p_2}$ = `intersect(Line2, Plane1)`

The length of the adjoining line segment is simply the distance between those two endpoints:

`length` = $|\vec{p_2} - \vec{p_1}|$

6 Future work

Our current implementation works fine for "nicely behaved" shapes. However, there might be some problems with non-optimal scenarios. For the 3D intersection point connector, for example, the decision tree which point to use (the skew-lines-point, the midpoint, or the shortest-distance-point) can surely be tweaked so the *intersection-point-connector* produces even better results.

What's more problematic is the routine for creating new parent geometry for 3D shapes after a parallel split. Currently it works for not too complex geometry and if the split direction is parallel to the connection-facet. For arbitrary geometry and to cover all cases and all split directions, this routine has to be revised.

Other ideas to think about in future are things like volume preservation or mass preservation. Just have a look at figure 8 `(e)`, the volume of the connected shape has increased significantly compared to the original shape `(a)`. None of these considerations has been implemented yet and could be a topic for future extensions.

References

Badler, N. I., Phillips, C. B., and Webber, B. L. 1993. *Simulating humans: computer graphics animation and control*. Oxford University Press, Inc., New York, NY, USA.

Fiedler, S. 2009. Procedural human posing using cga grammars. Tech. rep.

Ilcik, M., Fiedler, S., Purgathofer, W., and Wimmer, M., 2010. Procedural skeletons: Kinematic extensions to cga-shape grammars, 5.

Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Van Gool, L. 2006. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, SIGGRAPH '06, 614–623.

Papula, L. 2007. *Mathematik für Ingenieure und Naturwissenschaftler Band 1*, 11th ed. Viehweg.

Ratner, P. 2003. *3-D Human Modeling and Animation*, 2nd ed. John Wiley & Sons, Inc., New York, NY, USA.

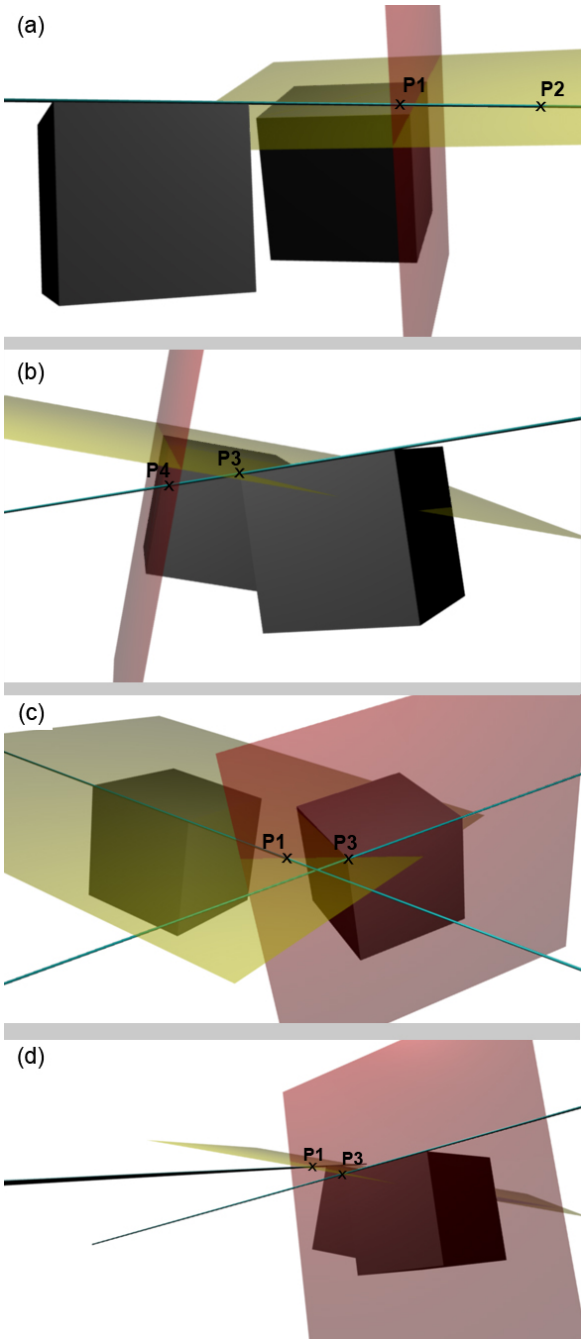

Figure 9: Under the assumption that the left shape is the parent shape and the right shape is the child shape, in figure (a) you see the parent's edge extended and intersected with the two planes adjacent to the vertex that are not the connect plane. Intersecting the edge with both planes yields two intersection points. P1, the intersection point with the red line, is chosen because it is located nearer to the original vertices. In (b) the same procedure is illustrated for the child's extended edge intersected with the parent planes. The intersection point with the yellow plane is nearer to the vertices than the one with the red plane, so P3 is chosen as the second connection-point candidate which is outlined in (c). Figure (d) shows the scene from a different perspective. It can be seen that the two extended edges don't intersect. They are skew lines.