3D building application for children

Problem presented by

Olga Timcenko

LEGO

LEGO is developing a 3D building application for children please see the current version on www.lego.com/ldd. We are now looking for possibilities to improve performances and functionality of the application. Classical LEGO bricks are building blocks which connect to each other in well-defined manner, and allow for making rigid structures of arbitrary shape (see Figure 1).

Figure 1: Typical rigidly-connected LEGO model

Recent development added different types of joints to bricks assortment (see Figure 2). So a typical LEGO construction consists of several rigidly connected blocks, connected together via different types of joints. (see Figure 3).

Unfortunately, as the construction is a result of a child s imagination, it is impossible to say anything about its structure kinematically, it could be anything,

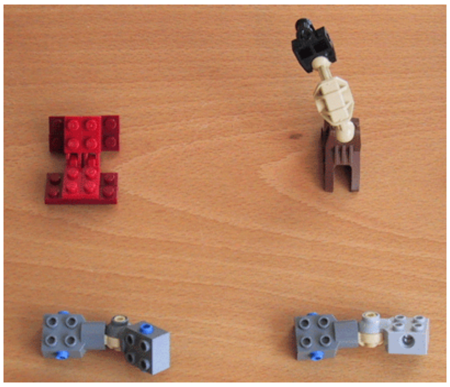

Figure 2: Typical LEGO joints

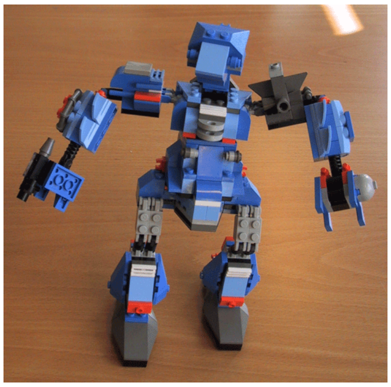

Figure 3: Typical construction with several rigidly connected parts connected via joints

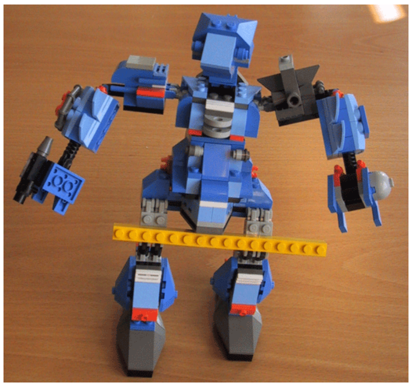

Figure 4: Changing kinematical structure of the previous model by adding a single brick

including both open chains and closed loops. Moreover, adding or deleting a single brick to a structure could completely change kinematical properties of the construction, as illustrated on Figure 4. The added/deleted brick could be very far from the joint itself, and not directly connected to any of the bricks forming the joint.

Computer game we are developing tries to mimic as faithfully as possible the process of building in real world.

What concerns us most is a data structure and set of algorithms that would allow us to update the structure of the digital model as fast as possible on-line i.e. a data structure that would allow for efficient adding/deleting a brick on-line (or with minimal delay for the user for the case of more complicated models) and would describe correct kinematical model at each moment as the child could decide at any moment that the model is finished, and then start adjusting joint positions, then continue building again, and so on. We would like to be able to build digital models with 200-500 bricks (current application significantly slows down at about 50 bricks).

Screen shot from existing application together with an example of a digital construction with several joints is on Figure 5.

Figure 5: Screen shot of the current application - open chains work fine, but there are some problems with parallel hinges

Study Group contributers

Sandra Allaare-Bruin, Morten Andersen, Thomas Bolander,
Paul Fischer, Peter Heres, Bryan Horton, Tom Høholdt,
Benny Lassen, Dana Mackey, and Henrik Gordon Petersen.
Report prepared by all of the above.

The LEGO study group. Unfortunately, Tom, Thomas and Paul are missing

1 Description of the problems

The LEGO problem was divided into several subproblems: Development of efficient datastructures is treated in Section 3. In Section 4, we handle the problem of improving speed for finding feasible docking points for a new brick. In Section 5 and 6, we handle different approaches for the automatic and efficient determination of whether a LEGO structure or substructure is rigid.

2 Notation used

We use in particular in Section 3 and 5 the following notation

- We let D be the dimension of the system (D=2 for planar systems, else D=3)
- N is the total number of subunits in the system, which have been determined to be rigid. We note arbitrary subunits by preferably i, j and k.
- We describe the position and orientation of the i'th rigid subunit by respectively the position $\boldsymbol{R}_i$ and orthonormal vectors $\boldsymbol{u}_{i1}$, $\boldsymbol{u}_{i2}$, $\boldsymbol{u}_{i3}$ (only the two first orthonormal vectors for planar systems).
- We note plug positions of the i'th rigid subunit by $\boldsymbol{p}_s^{(i)} \quad s = 1, \ldots, S_i$. Arbitrary plugs are numerated by s, t, u, v. The local coordinates of plug positions are written as $\boldsymbol{p}_s^{(i)} = \boldsymbol{R}_i + \sum_{e=1}^{D} X_{se}^{(i)} \boldsymbol{u}_{ie}$. If the plug can be used as a revolute joint, we denote the axis of rotation at the plug by $\boldsymbol{q}_s^{(i)}$. Local coordinates of the joint axis are $\boldsymbol{q}_s^{(i)} = \sum_{e=1}^{D} Y_{se}^{(i)} \boldsymbol{u}_{ie}$. Spherical joints need no axis.
- We denote the set of all $ND(D+1)$ coordinates by x.
- We let M be the total number of constraints in the system and index arbitrary constraints by greek letters, preferably $f_\alpha(x) = 0$,$f_\beta = 0$,$f_\gamma(x) = 0$.

3 Program and Data Structure

3.1 Program Structure

In order to determine what program functions are time critical a discussion of the response time expected by the user was done. The findings of this discussion are listed below. It should be noted here that the discussion group was rather small, so a more through investigation might be appropriate. For an explanation of what a connection point is see section 3.2

1 Add:

  - Search for compatible connection points - *Fast*
  - Update model - *Plenty of time*

2 Remove:

  - Visible response time - *Fast*
  - Updating the model - *Plenty of time*

3 Changing the camera position:

   - Visible response time - *Fast*
   - Updating the model - *Plenty of time*

4 Moving a hinge:

   - Collision detection - *Fast*
   - Visible response - *Fast*
   - Updating the model - *Plenty of time*

Conclusions:

- Fast:

  - Visible response
  - Search for compatible connection points
  - Collision detection

  The common reason why these program functions should be fast is that the user does something to the model and he or she should be able to see the effect of that almost immediately.

- Plenty of time:

  - Updating the model

  The reason why there is enough time to update the model in general is that the user has to choose something else (a brick, a tool, etc.) and while he or she is doing this there is plenty of time for the computer to update the model.

Computer graphics cards are today powerful enough to ensure that it is no problem to make the visible response time fast, so the main focus should be on making a data structure that enables the search for compatible connection points and collision detection to be fast. There is a fast and well developed method for collision detection based upon a hierarchic of oriented bounding boxes (OBB's), this will be discussed in section 4. For further comments about the search for compatible connection points see section 3.2.

3.2 Data structure

A Lego model is a collection of Lego bricks which can be connected in a multitude of different ways. A point on a Lego brick that can be connected to another Lego brick is called a connection point (see the next section for more detail). In order to get some structure on this a Lego model is split into:

- Rigid structures:
  A rigid structure is a collection of connected Lego bricks that, because of the constraints of the system, can only be translated or rotated as a whole (as long as no Lego bricks are added or removed).
- Connected hinges:
  A connected hinge is a collection of connection points that connects two different rigid structures with one or more degrees of freedom.

Determining whether or not a collection of connected Lego bricks is rigid will be discussed in section 4. The data structure of a rigid structure has to include:

- Data on connection points: This is used for adding and deleting bricks.
- Data on bounding boxes: This is used for collision detection (see section 4).
- Data on geometry: This is used for rendering.

And the data structure of a connected hinge has to include:

- Data on which rigid structures it connects.
- Data on which connection points are involved.

It should be noted the data structure presented in the following has been made without any thought about how to remove a brick from the Lego model. The reason why removing a brick has been neglected is that it was the impression that this was actually working with out any problems in the current implementation.

Connection point data

Assuming that it is possible for all connection structures (knobs, anti-knobs, etc.) to define a point, a direction, a axis of rotation (in case there is one degree of freedom) and a connection type such that they contain all the information necessary to determine how different rigid structures can be connected (this is to our knowledge the case for all the connection structures on Lego bricks used today). The only information needed, in addition to this, is knowledge about where the rigid structure is located in the global coordinate system (the coordinate system of the whole Lego model) and its orientation. This is then the information that the connection point data should contain.

Letting N denote the number of rigid structures, the connection point data of the i'th rigid structure will then be given as follows (this will at least be one way of representing it, there are definitely others). The representation of the connection point data will be exemplified with the simplified two dimensional brick shown in figure 6.

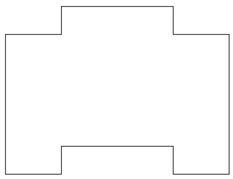

Figure 6: The two dimensional brick under investigation.

- Base point: $\boldsymbol{R}_i$.
  The base point of the rigid structure is an arbitrary but known point in the structure. The vector $\boldsymbol{R}_i$ is then the position vector of this point in the global coordinate system, i. e., $\boldsymbol{R}_i$ points from the origin of the global coordinate system to the base point of of the i'th rigid structure. This gives the position of the i'th rigid structure in the global coordinate system (see figure 7).

- Orientation (Local coordinate system): $\boldsymbol{u}_{i1}$, $\boldsymbol{u}_{i2}$, and $\boldsymbol{u}_{i3}$.
  The orientation of the rigid structure is given by the three vectors $\boldsymbol{u}_{i1}$, $\boldsymbol{u}_{i2}$, and $\boldsymbol{u}_{i3}$ with $\boldsymbol{R}_i$ as base point (see figure 7). They rotate and translate with the rigid structure. Using these as the basis vectors of the local coordinate system and writing the rest of the data structure with respect to this coordinate system insures that it is only necessary to rotate $\boldsymbol{u}_{i1}$, $\boldsymbol{u}_{i2}$, and $\boldsymbol{u}_{i3}$ and possibly translating the base point when the rigid structure is rotated.

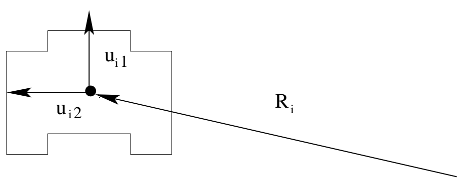

Figure 7: Base point and local coordinate system.

- Connection point vectors: $\boldsymbol{X}_s^{(i)}$
  The position vector in the local coordinate system of the s'th connection

point is given by the connection point vector $\boldsymbol{X}_s^{(i)}$, where $s = 1, \ldots, S_i$. S_i being the number of connection points of the i'th rigid structure. The s'th connection point is then in the global coordinate system given by

$$\boldsymbol{p}_s^{(i)} = \boldsymbol{R}_i + \sum_{e=1}^{3} X_{se}^{(i)} \boldsymbol{u}_{ie}.$$

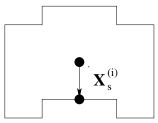

Figure 8: Connection point vectors.

- Connection vectors: $\boldsymbol{d}_s^{(i)}$
  In order to specify how two connection points should be oriented with respect to each other in order to fit together, the connection vector $\boldsymbol{d}_s^{(i)}$ is needed. This unit vector gives the direction of the connection in the local coordinate system. In order for two connections to fit together there connection vectors should be equal in the global coordinate system, where the connection vector in the global coordinate system is given by

$$\boldsymbol{w}_s^{(i)} = \boldsymbol{R}_i + \sum_{e=1}^{3} d_{se}^{(i)} \boldsymbol{u}_{ie}.$$

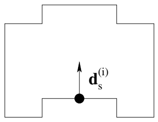

Figure 9: Connection vectors.

- Axis of rotation: $\boldsymbol{Y}_s^{(i)}$
  For the kind of connections where there is one degree of freedom the axis of rotation is also needed. This is represented by the unit vector $\boldsymbol{Y}_s^{(i)}$ in the local coordinate system (see figure 10) and in the global coordinate system it is given by

$$\boldsymbol{q}_s^{(i)} = \boldsymbol{R}_i + \sum_{e=1}^{3} Y_{se}^{(i)} \boldsymbol{u}_{ie}.$$

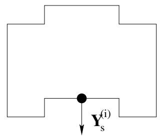

Figure 10: Axis of rotation.

- Connection type: $CT_i \in \{$Knob, Anti-knob, etc$\}$
  The last piece of information needed is the connection type. This is used to determine whether or not two connection points fit together. A Knob fit together with a Anti-knob for example.

Remark about the search for compatible connection points

One of the time critical program functions is the search for compatible connection points when adding a brick to the Lego model, i. e., a search for where a Lego brick can be added to the Lego model. So in order to make this fast it is important to reduce the set of connection points that the program has to search through. In the above representation of the data structure this can be done by either defining a connection type as occupied or by only including the connection points that are available for other Lego bricks. In addition to this it is a good idea to order the connection points according to how close they are to the Lego brick the user is trying to add and also whether the connection points are visible or not, i. e., whether they are in front of or behind the Lego model. This will be discussed in more detail in section 5.

4 Hierarchical Collision Detection

The problem of checking collisions between two objects has been studied quite intensively in the literature for applications within e.g. robotics and computer games. There exists very efficient methods based on hierachies of bounding sets. Two types of basic sets have been considered: Spheres and Oriented Bounding Boxes (OBB's). In [1], a hierachy of spheres is used and in [2], a hierachy of OBB's is used. Using spheres has the advantage, that it is efficient to compute wheter two spheres overlap, but the disadvantage that the spheres often do not give a tight bound of the object leading to unnecessary deep searches in the search tree. On the other hand, OBB trees are a little more expensive to check for overlap, but they bound the objects more efficiently. Experiences have shown that using OBB trees is for most applications to prefer. However, the method based on OBB trees is somewhat complicated to implement. In addition to these two methods, we give below a method that is not quite so efficient, but straightforward to implement and will give an improvement over the existing brute force method used by LEGO.

4.1 Problem Formulation

Given are n three-dimensional objects (bricks) B_i, $i = 0, \ldots, (n-1)$. and another object B_{new}. The task is, to check whether B_{new} intersects one of the B_i, $i = 0, \ldots, (n-1)$.

The brute force solution is to check B_{new} against all B_i, $i = 0, \ldots, (n-1)$. The time needed to check whether two objects B_{new} and B_i intersect is longer if the objects are more complex. For n checks this can be intolerably long. Therefore, one should single out a "few" objects B_i on which the full collision test is run.

4.2 First Step

For every geometrical object B_i, $i = 0, \ldots, (n-1)$ determine a small (as small as possible) sphere enclosing the whole object. Let c_i denote its center and r_i its radius. The radius is fixed once and for ever, the center depends on the current location of the object. Then one compares the distance between the two sphere centers $c_i = (x_i, y_i, z_i)$ and $c_{new} = (x_{new}, y_{new}, z_{new})$ and the sum of the radii.

$$\sqrt{(x_i - x_{new})^2 + (y_i - y_{new})^2 + (z_i - z_{new})^2} > (r_i + r_{new})$$

If this test is positive then the spheres around the objects do not intersect, hence the objects themselves do not. One might want to avoid the use of the square and use the following test instead

$$(x_i - x_{new})^2 + (y_i - y_{new})^2 + (z_i - z_{new})^2 > (r_i + r_{new})^2$$

Only those objects B_i not passing the test get into Step 2.

4.3 Second Step

For all remaining objects B_i and B_{new} compute an axis-aligned bounding. The box depends on the spacial orientation of the object and has to re-computed every time. Then the maximal (minimal) coordinates of B_{new} in every direction are compared the minimal (maximal) coordinates of B_i. Only those B_i whose bonding boxes overlap that of B_{new} in all directions get into step 3.

4.4 Third Step

Run the full collision test on the remaining B_i and B_{new}.

4.5 Remarks

For elongated objects (a 16 technic bar) one can use two or more small spheres. The object has to be enclosed in the union of them.

5 Improve speed of finding feasible docking positions

In the current version of the LEGO Digital Designer it could take a long time before the program decides where a new brick can be added to the already existing structure. This document serves to explain a decent way to quickly find the closest point where a new brick may be connected.

This chapter is composed as follows. In the next section the program characteritics and the current implementation of the docking procedure are drawn. This is followed by five sections, which are devoted to recommendations to make the docking procedure faster.

5.1 Program characteristics and current implementation

As was seen in the program structure the time-critical actions of the program are:

- Searching for compatible connection points
- Collision detection while moving rigid structures
- The visible response on whatever action made

Because the scene is pretty static, there is enough time to update the model. Hence, the solution strategy in this chapter will be to find fast ways to search through a static model.

The current version of the program considers a cone around the brick to be added. This cone has its top at the camera position and contains a certain sphere around the brick. All bricks of the structure which lie in this cone, and so has approximately the same 2D screen coordinates, are then considered. The docking position which is closest in 3D is taken as the candidate for connection: the brick to be added moves slightly to this position and turns from transparent into its normal color.
A disadvantage of this method is that all the connection points in this cone are still to be considered, both occupied and free, which can be quite a lot. This results in a slow adding procedure when the structure consists of 50 bricks. Currently it is not possible to handle structures which have more than 100 bricks. Furthermore,

it is possible at present to place bricks behind other objects, which is an unwanted feature. For very large structures it also apparent that the computer does not find a position, to place the new brick at all.

5.2 Projection of a 3D scene on a plane

Suppose we already have a built structure. The structure is viewed from a fixed camera position. This means that every 3D point is projected onto a 2D *image space*, see Figure 11. For every 3D-coordinate of the structure, it is very easy to

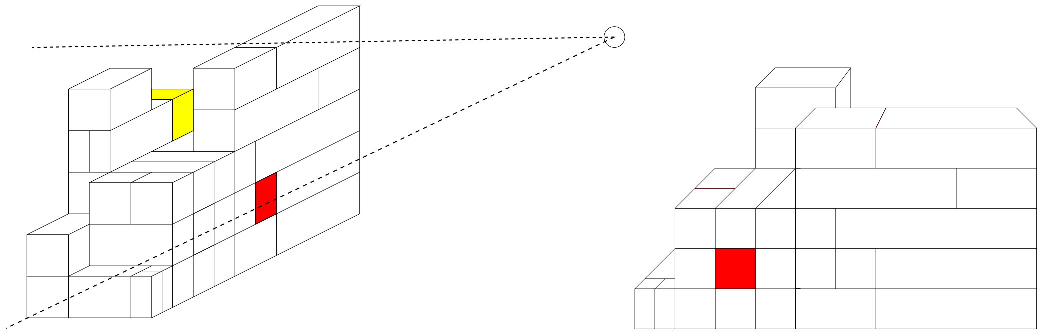

Figure 11: The camera position and the 2D image

calculate the 2D-screen coordinates. The three-dimensional volume under consideration is limited by a near-plane at $w = -n$, a far plane at $w = -f$ and a maximum viewing angle α. This leads to a truncated pyramid, see Figure 12. This viewing volume is projected onto some backplane with a distance d from the camera, the *image plane*. In Figure 12 the projection of the point (u, v, w) it taken as an example. The u-axis is pointed out of the paper in our direction. By similarity of triangles we have for the screen-coordinates (x, y) that:

$$x = \frac{ud}{w} \tag{1}$$

$$y = \frac{vd}{w} \tag{2}$$

It can be seen that u and v are always multiplied by a fixed constant. The z coordinate is always equal to $-d$, but we want to preserve the information of what is close to us and what is far away, so we will use the third coordinate to give us this information. We will continue using homogeneous coordinates, every 3D-coordinate (u, v, w) can be written in homogeneous coordinates as $(u, v, w, 1)$.

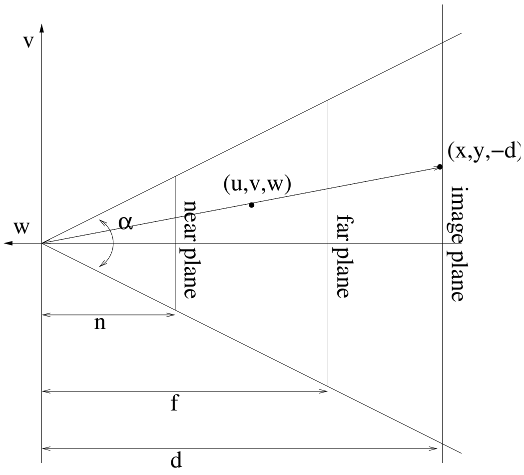

Figure 12: Definition of the viewing domain

The following operator projects the geometrical coordinates of $(u, v, w, 1)$ into the geometrical image space coordinates (x, y, z, s) with $-1 \leq x, y, z \leq 1$ [5]:

$$A = \begin{bmatrix} \cot(\frac{\alpha}{2}) & 0 & 0 & 0 \\ 0 & \cot(\frac{\alpha}{2}) & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3}$$

The operator defined here is invertible, so we can transform our image space back to the real 3D coordinates (u, v, w).

5.3 Viewing cone

Since the transformation described above is very cheap, we can calculate it ourselves, without making use of the graphics tool. Hence, we propose to consider only the connection points in a cone. These are all the available connection points with screen coordinates in the circle around the brick to be added.
To rigidly define what should be done, we define the cone that is considered. There exists a sphere in $\mathbf{R}^3$ with minimal radius in which the brick to be added is contained, see Figure 13. Associated with this there exists a cone in $\mathbf{R}^3$ with its top at the position of the camera and the minimal sphere enclosed in it:

5.4 Direction elimination

Another thing that could speed up the process is coming from back-face elimination [6]. There the information of the direction of a plane is used, to determine if

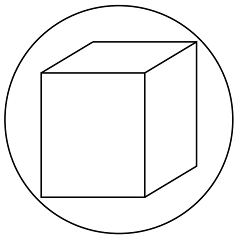

Figure 13: Sphere around brick to be added

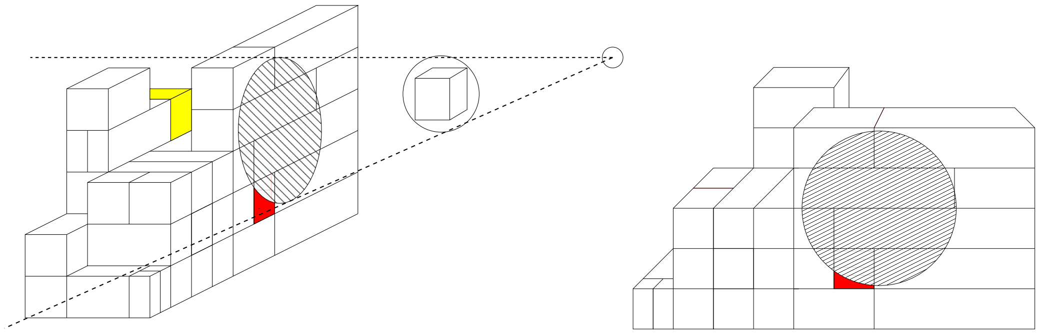

Figure 14: The viewing cone around the brick to be added

it has to be rendered or not. This is done by calculating the inner product between the viewing direction and the normal to the plane. If this is a negative number the plane should be drawn.
The idea can be used to divide all available connection points into groups according to their direction. Because the brick to be added can only be turned over a limited range (in the current implementation this is 60°) and therefore, groups of directions can be eliminated from consideration.

5.5 Visibility

When a 3D object is rendered, only the visible part has to be rendered. The same holds for adding a new brick to the structure; it is reasonable to say that a brick can only be added to visible connection points. In Figure 11 this would mean that the light grey brick at the backside of the structure is not available for connecting a new brick to. In some graphical tools, like OpenGL [3] or DirectX [4], information about visibility can be retrieved. This makes it possible to consider only the visible available connection points. A connection point is defined visible if a part of the connection point is visible. The list of visible available connection points is a subset of the available connection points and it changes every time the camera is moved.

In this way a small selection of possible connection points can be made. Once this is done the current algorithm for finding the exact docking place for connecting blocks can be applied as is done now.

OpenGL picking

In order to explain the picking algorithm, we assume that the LEGO program makes use of a graphical package like OpenGL. In OpenGL it is possible to assign numbers to individual objects in a name stack. One can return the numbers of the objects which are partially in a user defined region. This function of OpenGL is called *picking*. This function could be used to identify if an object is visible or not. If we define all knobs, anti-knobs, etc. being objects, we can determine if this object is visible: if there is a connection possible.

One method to reduce the amount of work done by the OpenGL pipeline during picking operations is to use a simplified form of the object in the picking computations. For example, individual objects can be replaced by geometry representing their bounding boxes. The accuracy of the picking operation is traded for increased speed. Some of the accuracy can be improved by adding a second pass in which the objects which are selected using their simplified geometry are reprocessed using their real geometry.

This picking feature can be used in two different ways:

- Every time the camera position changes, we update the list of all the visible available connection points in the scene.

- At the moment a brick to be added is moved around, the visible connection points are selected from the available connection points in the cone.

The method used depends on the speed of the picking algorithm. It should be fast, because the picking takes place every time the brick to be added is moved. Otherwise, the first method is a good alternative.

For many applications it may prove advantageous not to use the OpenGL pipeline at all to implement picking. For example, an application may choose to organize its geometric data spatially and use a hierarchy of bounding volumes to efficiently prune portions of the scene without testing each individual object.

5.6 Geometrical division of the LEGO scene in Octrees

The idea is to subdivide the scene into geometrically based cubes containing some of the objects of the scene. As was mentioned before searching for places to connect a rigid structure and collision-detection while moving some bricks must be very fast. So the idea is to maintain a model at all times, that makes these two actions very fast.

A very commonly used method of keeping track of a big scene in 3D-computer-games is to divide the scene into an Octree. Search google for ”octrees” to get access to some very good introductions to octree's.

The idea is to make a big cube containing the whole scene and then divide this into 8 smaller cubes by adding planes that divides the parent cube in half. This is done recursively for every cube containing more than one object or until a upper limit of recursive steps has been reaches or a maximum number of cubes has been reached.

In this case we can say that every brick is an object, or maybe the smaller atomic geometrical parts of the bricks are the objects, so that the knobs are considered being an object and a planar side of a brick is an object. If we use the latter approach a normal 2 by 1 brick (see Figure 15) can be divided into the following object:

- 2 knobs
- 2 anti knobs (below)
- 5 sides

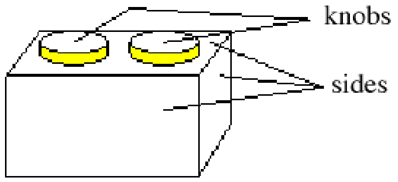

Figure 15: A 2 by 1 brick.

What approach that is chosen does not seem to be that important. The most important gain from this approach is that given a point or a line in space you can very fast eliminate all the cubes that are not containing that point or line. Let's try to illustrate the "search for a place to connect the brick" action in the 2-dimensional case.

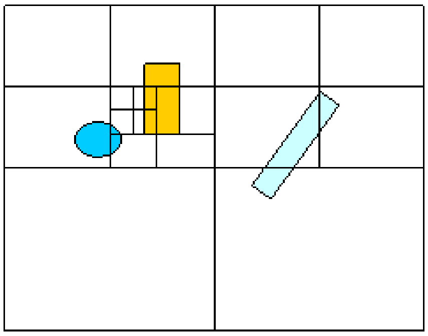

Figure 16: Partition of a two dimensional area

The scene containing two rectangles and a sphere is divided into cubes until every cube contains at most 1 object or until an upper limit of number of cubes is reached (see figure 16). In this case we do not reach this limit. All the cubes are added to a tree where the biggest cube contains the cubes that is positioned within that.

So, for this scene we get a tree with 25 cubes. Every cube holds references to its neighbors. These neighbors can be at the same level or one level up, which come down to at most 8 neighbors (see figure 17).

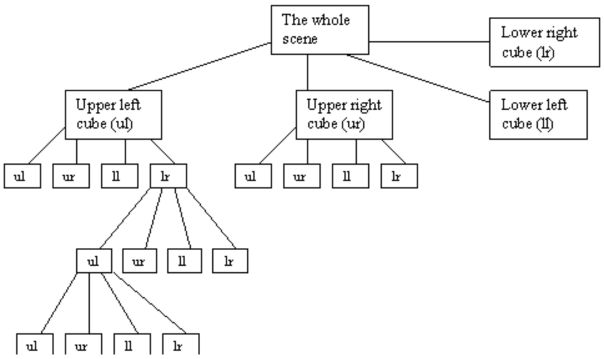

Figure 17: Octree

A search for a point in the octree (or here quadtree) is done by running through the tree from the root down figuring out what sub-cube the point belongs to using this recursive algorithm starting from the cube surrounding the whole cube.

1. For all the separating planes find out what side the point is on. All the planes are defined by a point in the plane P_0 plus a normal vector N. The way we can figure out whether a point in 3D space P is on one or another side of a plane, is by taking the inner product of the normal vector and the vector going in the direction from P_0 to P. If the result is positive then the point is on the same side as the direction of the normal vector, if the result is 0 then the point is placed on the plane, else it is placed on the other side of the plane.

2. If the resulting cube are subdivided into smaller cubes continue this algorithm. Else return the cube.

So for every level in the octree we have to do a few vector calculations which is very fast. So a guess is that for a tree with 10 level we have a maximum of 1.073.741.824 cubes (8 to the power of 10), which gives a nice detailed scene partition, with very few objects in every cube and the time it takes to search for a cube is 10 * 8 = 80 vector calculations.

Adding a brick

What we really need is to find the cubes that are on a line going perpendicular from screen projection all the way through the whole 3D scene. This can be done by finding the first cube that the line hits. This is done by finding the point where the line hits the first plane in the cube covering the whole scene and then find the smallest possible cube that contains this point. From this cube we find the neighbor that the line visits next. Since every cube has a list of its neighbors its possible to find the next cube from that list using the direction-vector and the starting point of the line simply by finding what separating plane the line reaches first on its way through the 3D space.

The neighbor is of the same size or bigger than the cube we leave, so in order to find the smallest possible cube we have to go downwards through the tree from the cube in focus.

Once we have found the next cube visited by the line it's time to check whether there is any connection points in the cube or whether the cube is totally covered by bricks so that there's no point in continuing.

If the cubes are very small it could be necessary to check for connection points in a radius around the line, so that not only the cubes containing the line but also a range of neighbors around it should be checked for possible connection points.

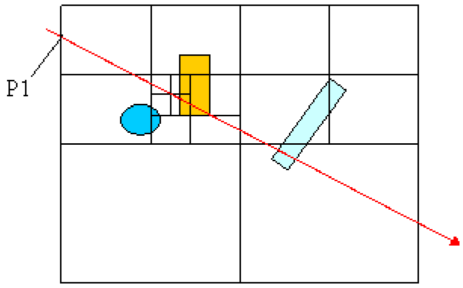

Figure 18: Line through a 2D example scene

Let us try to run through a visual example in 2D, where we find the cubes containing a line through the scene (see figure 18).

First we find the point where the line hits the outermost left plane (P_1). This point is within the upper leftmost cube, so that is checked for connection points. It is empty so there are no connection points and the cube is not covered totally, so we continue our search through space along the line. We find the neighbors of the upper left cube in the direction of the line by finding the first plane hit by the line and moving our attention to the cube on the other side of that plane. Since every cube holds a reference to its neighbors this cube can be found very fast.

If the cube is divided into smaller cubes we search down through the tree to find the smallest possible cube surrounding the line. This is continued until the line has passed all the way through the scene.

This will result in a list of cubes containing the line. Each of these contains a limited number of objects. Which makes the search for available connection points within the cubes found quite fast.

LEGO bricks, connection points and object size

How do we define a connection point?

Two LEGO bricks are connected if and only if they touch each other at least 3 places so that one of them can move in only 1 direction and only by applying a minimum amount of force.

If a plane on one brick touches another brick it is impossible for the other brick to move in a direction pointing towards that plane. Calling N the normal vector of plane pointing towards the brick it touches, we can say that movement in the direction v is only allowed if $N \cdot v \geq 0$.

We have the following LEGO-connection objects:

- Knob. Solid and with: diameter = 10 LEGO units
- Anti-knob. Not solid and with an inner diameter of 10 LEGO units

- Planes

We have the following types of connections:

- knob - anti-knob (see figure 19)

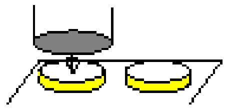

Figure 19: The knob - anti-knob connection type

- 3 knobs in the same plane pointing in the same direction in a square with side-lengths = 10 LEGO units and an anti-knob in the middle (see figure 20).

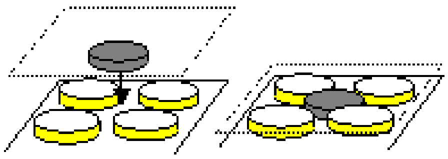

Figure 20: An anti-knob in the middle

- As above, but this time with a plane with width=5 LEGO units (see figure 21).

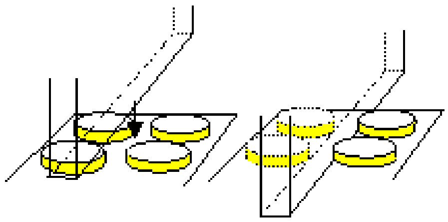

Figure 21: A side in between two knobs

- There could be more types. (There probably is), but the example above gives enough information to exemplify the proposed solution.

So to search for the available connection points we simply have to list all the:

- Knobs

- Anti-knobs
- 3 knobs in the same plane with distance = 10 LEGO units
- Planes with width = 5 LEGO units

This should be done so that each cube in the octree holds a list for each of the connection types.

As we want to find available connection points for one particular LEGO structure we find the connection points on that and for each of these find the nearest connection points that matches this type of connection point using the search for a line through 3D space algorithm listed above.

From this list of matching connection points we check whether connection makes a collision detection. If that is the case, the connection point is removed from the list of available connection possibilities, else it is chosen.

Local changes

As a brick / structure is added to a LEGO structure the changes are very local geometrically. This means that updating the octree can be done very efficiently and fast, since it is very few cubes that must be updated. The cubes are located nearby, so by using the fast algorithm for finding neighbors of cubes we can find the cubes very fast. The search for available connection points is also very fast. The user drags a brick over the 2D projection. The cubes to search through can be found by first finding the cubes on the line from the 2D projection through the 3D scene and then for all user-movements of the brick, to find the new cubes by navigating to the neighbors of the cubes. Collision detection can be performed using the cubes to find objects, that might collide, so as a structure is moved, the cubes where the structure is moved to are investigated for collision between objects in the structure moved and objects in the 3D scene.

If we make the cubes of a size, so that each hold at most 15 objects and so that the octree has at most 10 levels. The different actions take approximately:

- Find the cube for a point in 3D:
  - $< 10 * 8 = 80$ vector calculations.
- Find the next neighbor cube in a line though 3D:
  - 3 times the time it takes to find the point where a line goes through a plane ~ 9 vector calculations plus the time to go to the lowest level possible from the neighbor found < 80 vector calculations.
- Find the cubes on a line through 3D:

- The maximum number of cubes in one row of a scene divided into an octree with 10 levels is $210 = 1024$ cubes. The line can at most hit 2 times that $= 2048$ cubes. So in order to find a line through 3D, we have to find a point in 3D (< 80 vector calculations) and perform at most 2048 neighbor searches ($< 2048 * 9 = 18432$ vector calculations).
    - If there is a need to find lower levels of cubes it means that the cube we are in now is bigger, so time spent on finding smaller cubes are saved before because the cube that we started with is big. Therefore we can leave out the time spent on finding smaller cubes.
- This all sums up to:
    - Find point: 80 vector calculations
    - Find line through neighbor search: 18432 vector calculations
- This is a worst case, so practically it might be fast enough.

• Collision detection

- This can be done by:
    - Make a bounding box around the moving structure and then find all the cubes within that.
    - For all the cubes find the objects within these.
    - If there are collisions between the bounding box and the objects within the cubes, the actual objects in the structure moved are compared with the objects in the cubes.
- In most cases the number of objects within the cubes are small. So this algorithm should be sufficiently fast.

5.7 Recommendations

Our recommendations to speed up the process of finding possible docking positions are:

- Only consider the available connection points.
  Given the data structure a list of all available connection points can be generated easily. This is a list of all connection points in the built structure which are not occupied.

- Only consider the available connection points within the cone, around the brick to be added.

These are all the available connections points of which the screen coordinates lie in the sphere around the screen coordinates of the brick to be added.

- Only consider the groups of available connection points in approximately the right direction.
  Group the connection points according to their direction. Directions pointing away from the camera do not need to be considered.

- Only consider the visible available connection points.
  A picking algorithm can be used to find the visible connection points in a quick way.

- A geometrical hierarchy can be used to speed up the searching process. Octrees are a suitable choice to divide the geometrical domain.

6 Determining rigid substructures

6.1 Rigidity of Truss Structures

We will follow [7] and consider structures T that can be modelled by graphs $G(T)$ such that the joints correspond to the points of $G(T)$ and two points are adjacent if and only if there is a rod in T between the corresponding joints.
We assume that the rods are perfectly rigid and the joints are perfectly rotable.
Let$v$ denote the number of points and e the number of edges.
Suppose the joints $P_1, P_2, \dots$ have coordinates $(x_1, y_1, z_1), (x_2, y_2, z_2), \dots$ respectively, so we are working in 3D.
Let $A = (a_{kl})$ be the $e \times 3v$ matrix with

$$a_{kl} = \begin{cases} x_i - x_j & \text{if rod } k \text{ is between joints } i \text{ and} j\text{, and } l = i \\ y_i - y_j & \text{if rod } k \text{ is betweeen joints } i \text{ and } j\text{, and } l = i + v \\ z_i - z_j & \text{if rod } k \text{ is between joints } i \text{ and } j\text{, and } l = i + 2v \\ 0 & \text{otherwise} \end{cases}$$

Definition 1 *A 3D framework is rigid if* $rank(A) = 3v - 6$, *and a 2D framework is rigid if* $rank(A) = 2v - 3$.

Since $rank(A) \leq e$ we get in the 2D case:

1. *If the framework is rigid then* $e \geq 2v - 3$.

This condition is necessary but not sufficient, see later (.....should we give examples?—-)
.

This means that if we know the positions of the joints we can determine rigidity by linear algebra, however if we only know the graph we can not determine the matrix A, or even rank(A).
On the other hand if one suppose that the framework is *generic* meaning that the coordinates of the joints are algebraic independent over the rational numbers $\mathbb{Q}$, then the following theorem has been proved in [8].

Theorem 1 *If a 2D framework T with v joints and $e = 2v - 3$ rods is generic and G denotes the graph of T then T is rigid if and only if the edge set of G_x , obtained from G by doubling and edge x of G, can be covered by two edge disjoint spanning trees for every x.*

The nice thing is now that there is an efficient algorithm by Tarjan [9] for finding two edge- disjoint spanning trees,if they exist with complexity $O(vlogv + e)$ and by using that e times rigidity can be determined.

For the 3D case nothing like the above seems to be known.

6.2 Using constraint dynamics to describe general 2D and 3D structures

Consider first a planar system consisting of N rigid subunits. We then describe the configuration of the system using the $6N$ dependent coordinates $x \equiv (\boldsymbol{R}_i, \boldsymbol{u}_{i1}, \boldsymbol{u}_{i2} \quad i = 1, \ldots, N)$.

Definition 2 *A planar system is said to be* **rigid** *at the point x if it has only the three trivial degrees of freedom at x, namely translation of the whole system in the two directions in the plane and a rotation in the plane of the whole system.*

As mentioned, the coordinates are dependent. The coordinates $\boldsymbol{u}_{i1}, \boldsymbol{u}_{i2}$ of each subunit i are subject to 3 orthonormality constraints

$$\boldsymbol{u}_{ie} \cdot \boldsymbol{u}_{if} = \delta_{ef} \quad 1 \leq e \leq f \leq 2 \tag{4}$$

where δ_{ef} is the socalled Kronecker δ defined by $\delta_{ef} = 1$ if $e = f$ and zero otherwise.

If the rigid subunits i and j are joined together at the point $\boldsymbol{p}_s^{(i)} = \boldsymbol{R}_i + \sum_{e=1}^{D} X_{se}^{(i)} \boldsymbol{u}_{ie}$ on subunit i and on $\boldsymbol{p}_t^{(j)} = \boldsymbol{R}_j + \sum_{e=1}^{D} X_{te}^{(j)} \boldsymbol{u}_{je}$ on subunit j, we

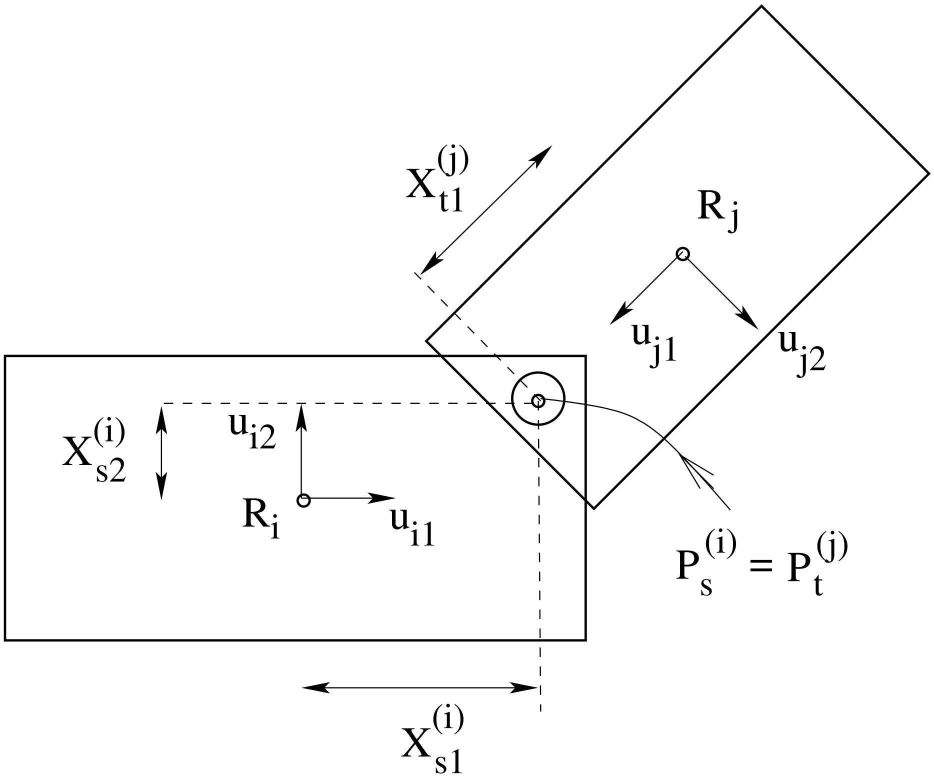

Figure 22: Constraints due to revolute joint in 2D

get furthermore two constraints (see Figure 22) $\boldsymbol{p}_s^{(i)} - \boldsymbol{p}_t^{(j)} = \mathbf{0}$ or described using our coordinates

$$\boldsymbol{R}_i + \sum_{e=1}^{D} X_{se}^{(i)} \boldsymbol{u}_{ie} - \boldsymbol{R}_j + \sum_{e=1}^{D} X_{te}^{(j)} \boldsymbol{u}_{je} = \mathbf{0} \tag{5}$$

In the following, we let $x \in \mathbb{R}^{6N}$ denote an arbitrary set of coordinates satisfying all the constraints.

Definition 3 *A set of constraints* $f_\alpha(x) \quad \alpha = 1, \ldots, M$ *are called* **independent** *at the point* x *if the set* $\{\nabla f_1(x), \ldots, \nabla f_M(x)\}$ *is linear independent where* $\nabla f_\alpha(x)$ *is the gradient of* $f_\alpha(x)$ *with respect to* x*.*

A planar system with N rigid subunits and J joints thus is described by $6N$ coordinates and $3N + 2J$ constraints. It is well known from classical mechanics[10] that the system will be rigid at the point x if and only if

$$6N - M = 3 \tag{6}$$

where M is the number of independent constraints at x. (For readers not familiar with classical mechanics, this can be seen by considering a virtual infinitely small displacement δx away from x. This displacement must satisfy the M independent linear equations given by the differential of the constraints $\nabla f_\alpha(x)\delta x = 0 \quad \alpha = 1, \ldots, M$. This leaves us with only the three independent parameters of x which must correspond to translation and rotation of the whole system.)

We know that $M \leq 3N + 2J$ and thus obtain a simple *necessary* condition for a system to be rigid at some configuration

First necessary condition for rigidity of a structure: Consider a planar structure with N rigid subunits and j joints. If the structure is rigid at some configuration x, it satisfies the inequality

$$3N - 2J \leq 3 \tag{7}$$

This condition can be used in a search procedure for rigid structures together with a second necessary condition

Second necessary condition for rigidity of a structure: Consider a planar structure with N rigid subunits and j joints. If the structure is rigid at some configuration x, every rigid subunit is joined to at least two other rigid subunits.

The second condition simply means that we can of course have no open ends in a rigid structure. In the first step of a general search for rigid substructures, we can search for structures satisfying the two necessary conditions. Having found a candidate structure satisfying these two conditions, we should consider the independency of the constraints more closely.

If we have candidates for rigid structures, we can check condition 6 in a standard procedure for checking linear independence. This procedure has an asymptotic computational overhead $\mathcal{O}(M^2N)$. We have however no polynomial time algorithm for finding all candidate structures except in the case described in the previous section, where the overall structure can be mapped as a 2D truss structure.

The situation in $3D$ is more or less similar. Here we have $12N$ dependent coordinates $x \equiv (\boldsymbol{R}_i, \boldsymbol{u}_{i1}, \boldsymbol{u}_{i2}, \boldsymbol{u}_{i3} \quad i = 1, \ldots, N)$.

Definition 4 *A spatial system is said to be* **rigid** *at the point x if it has only the six trivial degrees of freedom at x, namely translation of the whole system in the three directions in the plane and rotations around three fixed orthonormal axes.*

The coordinates are subject to 6 orthonormality constraints

$$\boldsymbol{u}_{ie} \cdot \boldsymbol{u}_{if} = \delta_{ef} \quad 1 \leq e \leq f \leq 3 \tag{8}$$

If the rigid subunits i and j are joined together with a spherical joint at the point $\boldsymbol{p}_s^{(i)} = \boldsymbol{R}_i + \sum_{e=1}^{D} X_{se}^{(i)} \boldsymbol{u}_{ie}$ on subunit i and on $\boldsymbol{p}_t^{(j)} = \boldsymbol{R}_j + \sum_{e=1}^{D} X_{te}^{(j)} \boldsymbol{u}_{je}$ on subunit j, we get three constraints as before $\boldsymbol{p}_s^{(i)} - \boldsymbol{p}_t^{(j)} = \boldsymbol{0}$ or described using our coordinates exactly as before

$$\boldsymbol{R}_i + \sum_{e=1}^{D} X_{se}^{(i)} \boldsymbol{u}_{ie} - \boldsymbol{R}_j + \sum_{e=1}^{D} X_{te}^{(j)} \boldsymbol{u}_{je} = \boldsymbol{0} \tag{9}$$

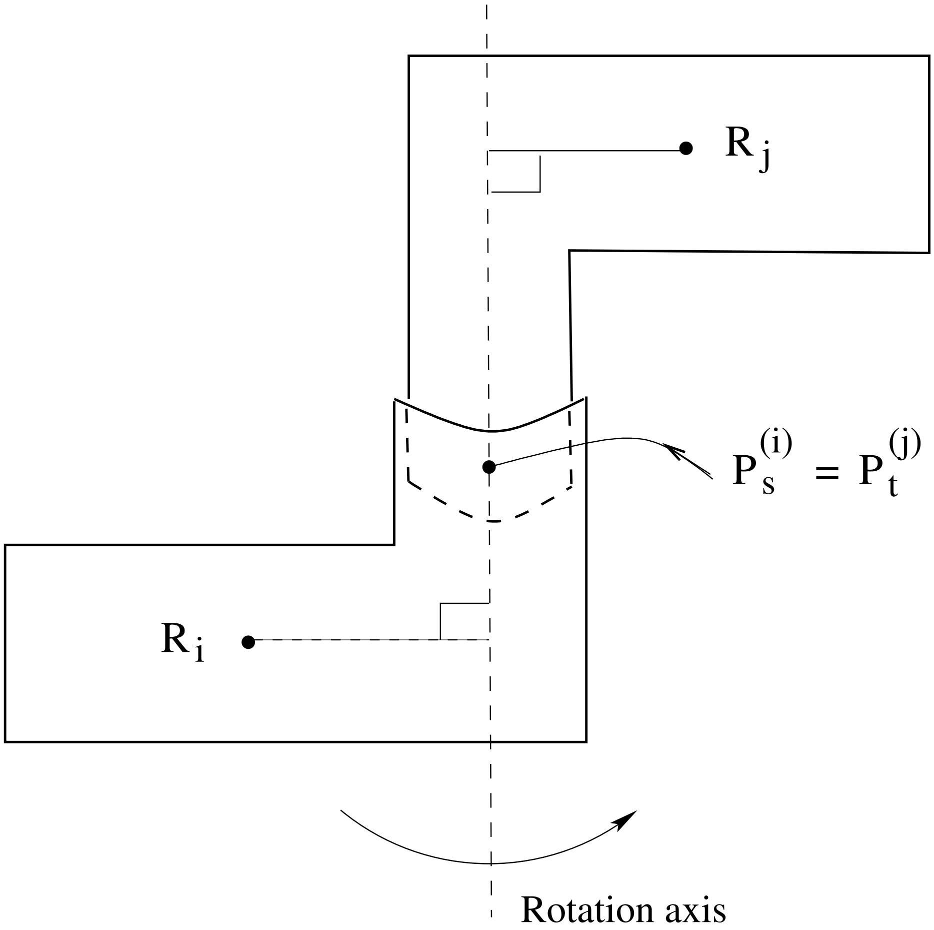

Figure 23: Constraints due to revolute joint in 3D

However, if the joint is revolute, the coordinates are subject to two additional constraints. Let the axis of rotation be described in local coordinates as the constant unit vectors $\boldsymbol{q}_s^{(i)} = \sum_{e=1}^{D} Y_{se}^{(j)} \boldsymbol{u}_{ie}$ and $\boldsymbol{q}_t^{(j)} = \sum_{e=1}^{D} Y_{te}^{(j)} \boldsymbol{u}_{je}$ respectively, we can write the additional constraints as (see Figure 23)

$$\{(\boldsymbol{R}_i - (\boldsymbol{R}_i \cdot \boldsymbol{q}_i)\boldsymbol{q}_i\} \cdot \boldsymbol{q}_j = 0 \tag{10}$$

$$\left\{(\boldsymbol{R}_j - (\boldsymbol{R}_j \cdot \boldsymbol{q}_j)\boldsymbol{q}_j\right\} \cdot \boldsymbol{q}_i = 0 \tag{11}$$

where we for analysis purposes assume that we have chosen the representative points $\boldsymbol{R}_i$ to not lie on any joint axis of the corresponding rigid subunit.

We thus get $6N+5J$ constraints and a necessary condition $6N-5J \leq 6$. There is unfortunately no known polynomial-time algorithm for finding substructures satisfying the necessary condition not even for $3D$ truss structures.

6.3 Two platform structures

We can however solve a special case completely, namely the case of two platform structures (see Figure 24). A two platform structure is a closed structure, where all rigid subunits except the two platforms have exactly two joints. Qualitatively one can think of two platforms connected to each other through a number of multi-joint limbs.

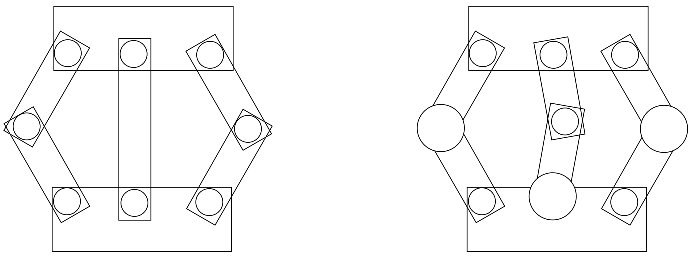

Figure 24: Two platform structures (left picture in 2D, right picture in 3D). Big circles illustrate spherical joints.

Consider first the $2D$ case. Here we have the necessary condition $3N - 2J \leq 3$. We thus get the following complete set of candidates for rigid structures

$$\text{2 limbs:} \quad (1,2) \tag{12}$$

$$\text{3 limbs:} \quad (2,2,2) \tag{13}$$

where e.g. $(1,2)$ means one limb with one 1DOF joint and one limb with two 1DOF joints. Similarly, we get in the $3D$ case the necessary condition $6N-5J \leq 6$. We then get structures

$$
\begin{array}{lcr}
\text{2 limbs:} & (3,3) & (14) \\
\text{3 limbs:} & (3,4,5) & (15) \\
\text{4 limbs:} & (3,5,5,5);(4,4,4,6);(4,4,5,5) & (16) \\
\text{5 limbs:} & (4,5,5,5,5) & (17) \\
\text{6 limbs:} & (5,5,5,5,5,5) & (18)
\end{array}
$$

6.4 $2\frac{1}{2}$D structures

Unfortunately, there are structures, which are neither purely $2D$ nor purely $3D$. We call them $2\frac{1}{2}D$ structures. Examples of these structures are shown in Figure 25.

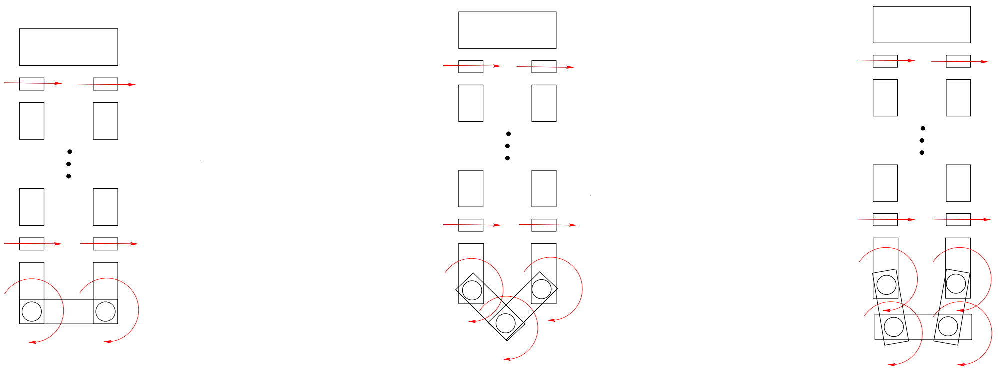

Figure 25: $2\frac{1}{2}D$ structures

The structure to the left often occurs in LEGO models. We have one loop of J joints in one direction and two joints in another direction. Then the substructure consisting of the 3 units at the two joints in the second direction will always be rigid. It thus reduces to a planar structure with J joints and $N=J$ links. In general, this will be non-rigid if and only if $J \geq 4$. However, if there are colinear joints as in the figure, it can be shown that there will be one dependent constraint for each pair of colinear joints. For the structure to the left in Figure 25, we thus get $M = 3N + 2$ for $J = 2$ and $M = 3N + 2(J - 1)$ for $J \geq 4$ (notice that J is even). For $J = 2$, we thus get $6N - M = 4 \geq 3$. This structure is thus non-rigid. For $J \geq 4$, we get $6N - M = 3N - 2(J - 1) = J + 2 \geq 3$ and thus again non-rigid structures.

6.5 Determination of purely spatial mechanisms

A structure satisfying the second necessary condition for rigidity is purely spatial if and only if there for any closed loop in the structure exists two rigid subunits such that when considering only one of the two chains from the considered closed loop between them, one of the subunits can move in all six degrees of freedom with respect to the other one. Notice that this chain must contain at least six degrees of freedom. A way to determine whether this is true for any given subunits with a given chain between them is to check whether the corresponding "Manipulator Jacobian" known from robotics (see e.g. [11]) has rank 6.

6.6 Recommendations for future work

We have completely solved the problem for purely planar or purely spatial 2 platform structures. Moreover, we have a general method to determine if a structure is purely planar or purely spatial. However, we may have 2 platform structures that are neither purely spatial nor purely planar. Some of these have already been shown in Section 6.4. However, one can also imagine other structures such as for example a situation where some of the loops are spatial and others are planar !!! What remains to be studied is

- To determine in general the "dimensionality" of a structure (this can be several numbers for mixed dimensionalities).
- To find inequalities such as those for the purely spatial and purely planar structures that give necessary conditions for the structure to be rigid.
- To use these inequalities to search for efficient methods for searching for candidates for rigid substructures.

7 Heuristic Approach to Rigidity Detection

7.1 Introduction

The LEGO Digital Designer is a program for constructing virtual LEGO models on a computer. The user can interactively add or delete bricks. Figure 26 shows an example. A model constructed in this way may be flexible. By using hinges,

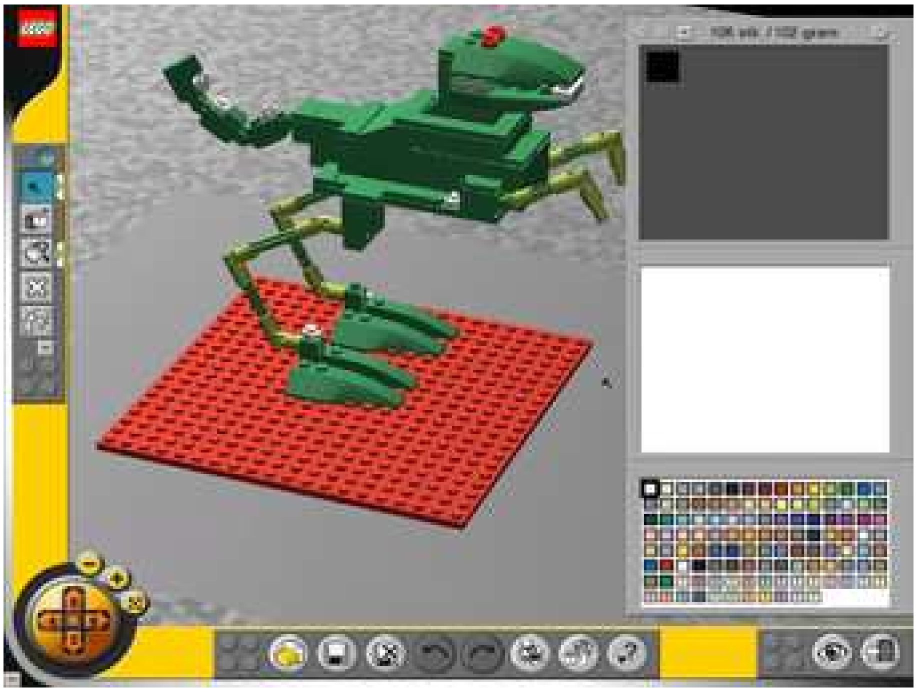

Figure 26: Screen shot of the LEGO Digital Designer.

parts of the model can be moved relative to others.

The problem is to

(i) Determine the (maximal) rigid substructures of a LEGO model.

(ii) Determine how these substructures can move relative to one another.

The point of solving these problems is to allow the constructor of a virtual LEGO model to move the various parts of the model on the screen in the same way that these parts would move in reality. In this report we will only explicitly consider problem (i), but our approach to solving it is influenced by the ambition for the solution to be fairly readily extensible into a solution of (ii).

The general problems of detecting the maximal rigid substructures of a truss structure are know computationally hard. We propose a heuristic algorithm for this problem, in the hope that LEGO models are not the "hard instances" for rigidity detection.

We will first introduce the basic concepts involved in the problem. We take the notion of when two (or more) bricks are *connected* to be given. There are two basic types of connections between pairs of bricks: *rigid* and *non-rigid*.

- A connection is *rigid* when the two bricks cannot be moved relative to one another without destroying the connection.

- A connection is *non-rigid* (or *flexible*) if the bricks can be moved relative to one another.

For standard LEGO bricks, the connection is non-rigid if the two bricks have only one connection point (knob) in common (Figure 27).

Figure 27: A simple non-rigid connection

If they have more than one connection point in common, it is a rigid connection (Figure 28). Generally, we assume it to be known when two bricks are

Figure 28: A simple rigid connection

connected rigidly and when they are connected non-rigidly.

A collection of connected bricks is called a *structure*. A structure is thus simply a (connected) LEGO model. Structures obtained by removing some of the bricks of a structure M are called *substructures* of M. A structure is called *rigid* if no substructure of it can be moved relative to some other substructure (without destroying any of the connections). In other words, if one brick of a rigid structure is moved, all bricks of that structure moves along. Obviously, if all connections of a structure are rigid, then the entire structure will be rigid. Given the above definitions, the notion of a *maximal rigid substructure* is defined in the obvious way. Note that rigid structures can contain flexible connections (see Figure 31 below).

The simplest rigid structures are those which do not contain any flexible connection. A maximal substructure without any flexible links is called a *superbrick*. A superbrick behaves as an ordinary (complex) brick in all respects apart from the fact that a superbrick can be split into smaller parts (bricks). All notions defined for bricks are inherited by superbricks, including the notions of when two superbricks are rigidly or non-rigidly connected. The two bricks in Figure 28 form a superbrick. It could of course just as well have been a single brick (and it would become one if we glued the two bricks together). Not all rigid structures are superbricks. Figure 31 gives an example of a rigid structure which contains two hinges and is thus not a superbrick.

For now, we will make a number of limiting assumptions concerning the bricks available. These assumptions will exclude a number of LEGO bricks that actually occur in some complex LEGO models, but the assumptions are necessary to make an initial simplification of the problem at hand. The assumptions are:

- No single LEGO brick is flexible, that is, it cannot be bend, twisted or elongated. When considering LEGO hinges, these are taken to consist of two individual non-flexible bricks connected non-rigidly.

- All non-rigid connections form one-dimensional hinges. That is, there is a *single* axis of rotation around which the two bricks can be rotated relative to each other. Such situations are illustrated in Figure 27 and 29. We thus exclude bricks with ball-links.

We propose to use graph models to describe LEGO structures. The model is hierarchically structured. The fine structure of a LEGO model is described by the *brick graph*. The *component graph* abstracts from the individual bricks and provides the information in the flexibility of the model. We consider the static situation in which a (maybe incomplete but fixed) structure is given. Later we shall address the case where the structure is dynamically changed by adding or removing bricks.

Brick graphs

Brick graphs contain the information on all individual connections, rigid and flexible, of bricks in the model. A brick graph has a node for every brick in the model. The edges correspond to connections. There are two types of edges:

Figure 29: A LEGO hinge

- *Rigid edges* drawn as solid lines: 

- *Flexible edges* drawn as dashed lines: 

If two bricks are connected, an edge of the appropriate type is added between the nodes.

The two types of edges are used at different levels in the modelling process. First only the rigid edges are considered to detect the superbricks. It is easy to see that the superbricks of a structure correspond to the maximal connected components of the brick graph, when only the rigid edges are considered. The superbricks then replace the bricks, thus simplifying the graph model.

Component graphs

The component graph is used to detect the maximal rigid substructures. It is dynamically changing in the computational process. The *component graph* is a labelled multi-graph (parallel edges allowed). The following properties remain invariant:

- The nodes correspond to (not necessarily maximal) rigid substructures. In the following we will often refer to these as *components*.

- The edges correspond to flexible connections (dashed lines). Every edge is labelled with the *axis of rotation* of the corresponding flexible connection. The axis of rotation is taken to be a straight line in space, so it carries information both about the orientation of the axis and of its position in space.

- For every flexible connection in the structure, there is an edge between the rigid components that it connects.

An example is shown in Figure 30.

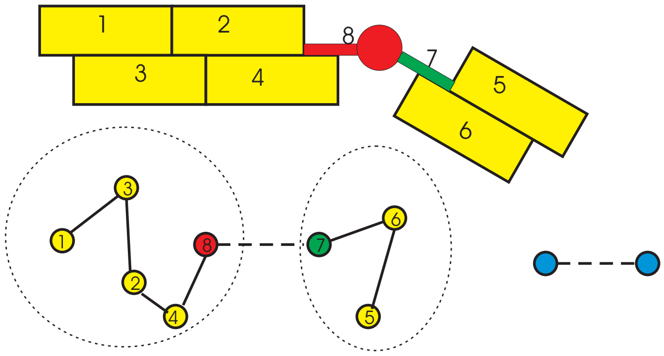

Figure 30: A LEGO model (top) and the induced brick graph (lower left) and the component graph (lower right). Labelling of dashed edges is omitted.

7.2 Constructing the Graphs

We assume that the following information on the physical LEGO model is available:

- The number of bricks of the model.
- The connection structure, i.e., which brick are connected to which.
- The type of the connection, rigid or non-rigid. For non-rigid connections, also the type of hinge as well as the position and orientation of its axis of rotation.

The brick graph is constructed by inspecting the physical model. Let N be the number of bricks and M be the number of connections. The time to build the brick graph is $O(N + M)$. Then the superbricks are found by a depth-first-search algorithm in time $O(N + M)$. [1] This algorithm only uses the rigid edges of the brick graph, ignoring the flexible ones.

When this is done, we can build the component graph. The graph is initialized as follows:

[1]For basic notions and results in algorithmics we refer to Cormen et al.: Introduction to Algorithms.

- For every superbrick in the structure, we put a corresponding node in the graph. Superbricks and their corresponding nodes will be identified in the following.

- For every flexible connection in the structure, we put an edge between the superbricks that it connects and label the edge with the axis of rotation of the connection. This is to be understood such that if there are, say, two non-rigid connections between a pair of superbricks, then there are also two (parallel) edges between the corresponding nodes. This information is derived from the brick graph.

In an implementation, the component graph would probably be initialized in the depth-first-search algorithm which determines the maximal connected components.

The component graph is then simplified to find the maximal rigid components. In this process nodes might be contracted (merged). Also a set of edges which represents co-linear hinges will be replaces by a single edge. We will now show how to analyse and simplify the component graph to find the maximal rigid substructures (henceforth MRSs). The MRSs are to be collected in a list L. The graph is simplified in a number of consecutive steps, and whenever a simplification step leads us to find a new MRS, we will add this to the list L. The individual steps of our *simplification algorithm* are sketched below.

Step 1: Merging connections

Consider all flexible edges in the component graph. Let G_{XY} be the group of all edges between node X and node Y. For each group check which of the following cases applies:

$X = Y$	Then do nothing.
$(X \neq Y) \wedge (\lvert G_{XY}\rvert = 1)$	Then do nothing.
$(X \neq Y) \wedge (\lvert G_{XY}\rvert > 1)$	If all flexible connections in G_{XY} are co-linear, i.e. all labels in G_{XY} are identical, replace the edges in G_{XY} by a single edge. If, conversely, there is a pair of non-identical labels in G_{XY}, then contract the two nodes X and Y to a single node. [2]

Figure 31 shows an example.

[2]For a definition of ‘contraction’ and other basic notions in graph theory we refer to Béla Bollobás: Modern graph theory.

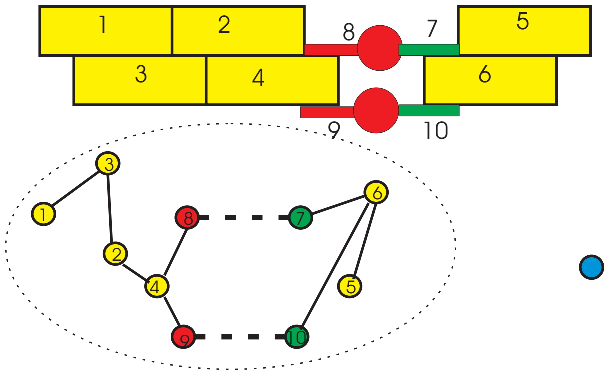

Figure 31: Two hinges with non-collinear axes form a rigid connection. The single node to the right is the component graph after step 1 has been completed.

Step 2: Acyclic nodes

When a node in a graph is not contained in any cycles we call it an *acyclic node*. Consider two adjacent nodes p and q of the component graph. If the edge connecting p and q is the only path from p to q then the corresponding substructures are non-rigidly connected (since the edge between them represents a flexible connection). From this it follows that all acyclic nodes are only involved in non-rigid connections. In other words, all substructures corresponding to acyclic nodes constitute maximal rigid substructures. We can therefore remove them from the graph and put them into the list L of MRSs. This can all be completed in linear time: We repeatedly find and remove all maximal paths starting in nodes of degree 1 and containing only nodes of degree ≤ 2. When step 2 is completed, all nodes of the component graph are cyclic (contained in a cycle).

Step 3: Cycles of length 3

Note, that cycles of length 2 contain exactly two nodes. They are thus covered by Step 1 above. Cycles of length 3 may or may not form rigid structures. Figure 32 shows examples for both. To check which situation we are in, we look at the labels of the edges, i.e., the axes of rotation.

After every simplification, Steps 1 through 3 are repeated until no further simplification is possible.

Step 4: Structures with at least 4 nodes

For structures with 4 or more nodes we propose to build a library in which labelled component graphs and their simplifications are stored. An example can be seen

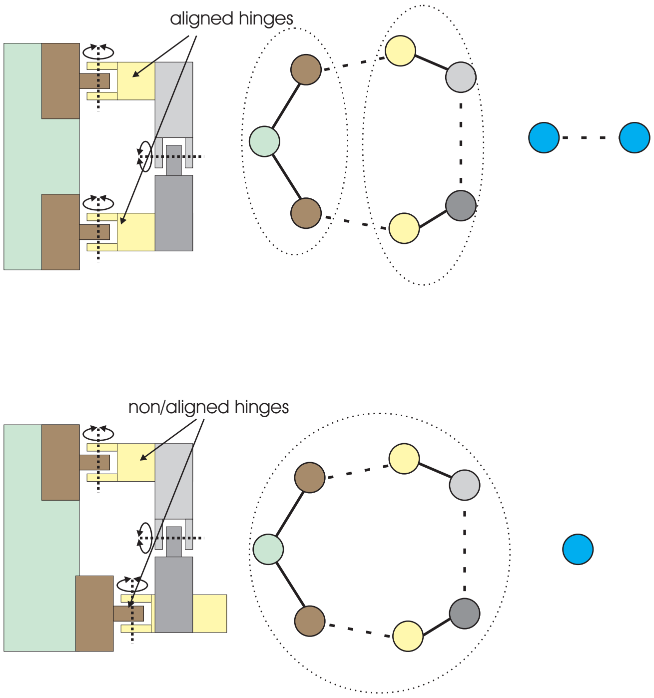

Figure 32: An example for non-flexible hinges. The left column shows the physical arrangement, the middle column shows the brick graph, and the component graph after the completion of step 3 is drawn at the right. The example in the first row has three superbricks but only two MRSs. The hinge at the right can never be moved. In the lower row only one MRS is present because the hinges at the left are no longer co-linear.

in Figure 33. If the component graph contains a subgraph, which is stored in the library, then it is replaced by its simplification.

Again, after every simplification, Steps 1 through 4 are repeated until no further simplification is possible.

7.3 Work plan

- Specification of the graphs, e.g., what information has to be stored at the nodes and edges.
- Building the library: Detection of (frequently occurring) substructures. Analysis of these substructures with respect to possible simplifications.
- Finding a appropriate representation of substructures which allows fast search

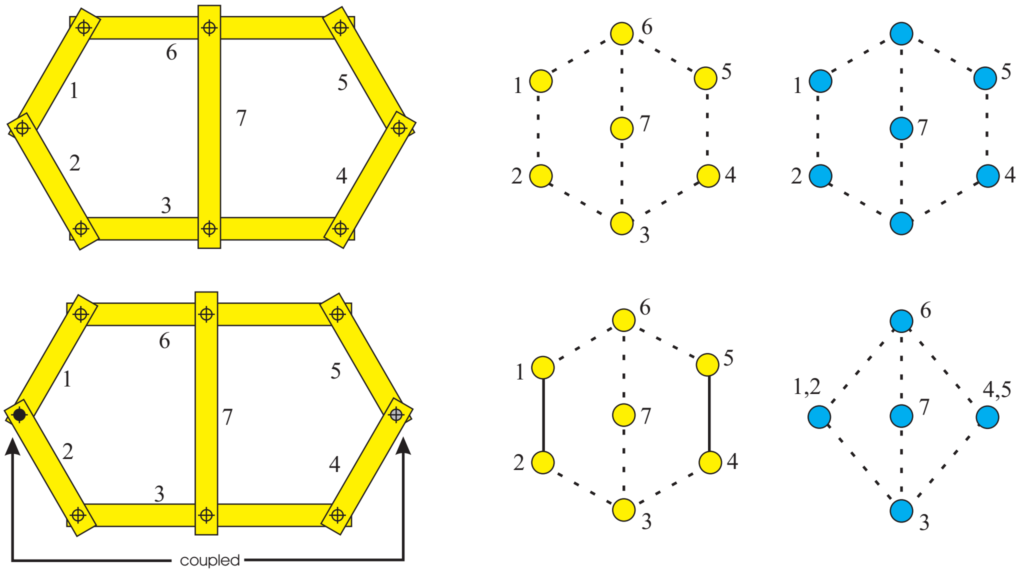

Figure 33: An example for indirect coupling of flexible connections. An arrangement of seven LEGO bars connected by eight hinges. The left column shows the physical arrangement, the middle column shows the brick graph, and the component graph is drawn at the right. In the upper row, the hinges between bars 1, 2 and bars 4, 5 can be moved. If the hinge between bars 1 and 2 is made rigid (lower row), then the hinge between bars 4 and 5 becomes rigid as well. This happens even though bars 1 or 2 do not have any direct connection to bars 4 or 5.

in the library. A "signature" representation might be promising.

- Development of a (may be interactive tool) for analysing substructures with respect to rigidity and possible simplifications.
- Analysis of the simplification algorithm for the component graph. Is the repetition of all previous steps really necessary after every replacement? Are there situations which require a different treatment.
- Which effect does the order of simplifications have.
- Time analysis of the simplification algorithm.
- Implementation issues: Recommended data structures. Efficient implementation of the parts of the algorithms, etc.
- Making the data structures fully dynamic: Additions and deletions of bricks

are allowed. Find efficient ways to update the brick graph and the component graph.

- Analysis of other approaches than the combinatorial one:
  - Algebraic representation: Angles as a system of quadratic equations. A structure is non-rigid if there is a connected solution set. Rigid if only isolated solutions exist.
  - Logic representation: As a set of formulas. Which structures correspond to solvable formulas (Horn clauses)?

References

[1] S. Quinlan: *Efficient Distance Computation between Non-Convex Objects*. In Proc. of the IEEE Int. Conf. on Robotics and Automation, 1994

[2] S. Gottschalk, M.C. Lin and D. Manocha: OBBTree: *A Hierarchical Structure for Rapid Interference Detection* Computer Graphics (1996)

[3] www.opengl.org

[4] www.microsoft.com/directx

[5] Kenneth. I. Joy, *The Viewing Transformation*, On-line Computer Graphics Notes, available on `http://graphics.cs.ucdavis.edu/GraphicsNotes/Viewing-Transformation.pdf`

[6] Alan Watt and Fabio Policarpo, *3D Games*, Pearson International, 2000

[7] Andras Recski *Matroid Theory and its Applications* Sections 6.3 and 12.3 Springer 1989

[8] L. Lovasz and Y. Yemeni: On Generic Rigidity in the Plane *SIAM J. Alge. Disc. Meth.* Vol 3 no.1 March 1982 pp. 91-98

[9] R.E.Tarjan: Edge-disjoint spanning trees, dominators and depth-first search. STAN-CS-74-455 Stanford University September 1974.

[10] See e.g. H. Goldstein: *Classical Mechanics*, ISBN 0-201-02918-9, (Addison-Wesley)

[11] R. J. Schilling, *Fundamentals of Robotics*, ISBN 0-13-344433-3 (Prentice-Hall)