Linköping University
2005-01-13

SPLITTER

Simulating the Breaking of Glass in Real Time

Jesper Carlson, jesca144@student.liu.se
Daniel Enetoft, danen736@student.liu.se
Anders Fjeldstad, andfj645@student.liu.se
Kristofer Gärdeborg, kriga592@student.liu.se

Abstract

The goal of the Splitter project was to simulate the breaking of glass window panes in a graphical application in real time with some degree of realistic physics. We accomplished this by starting from what is known about the different types of crack patterns that can arise, working our way towards a simple and fast model that still produces reasonable-looking results. The simulation can be broken down into four main steps, namely:

1. Deciding if and how the glass will break.
2. Setting up the data structure that represents the pattern.
3. Generating the crack pattern.
4. Detecting all loose pieces that may have formed in the pattern.

Whether (and how) the glass breaks is in our model partly based on "real physics" but mostly on general knowledge of different possible crack patterns and their characteristics. This makes the process fast while not sacrifying too much realism, in our opinion.

The data structure we developed can be described as a web of connected Points, each having information about itself and its connection to other Points. There is also a separate list of all Points, kept for making access to Points faster in some cases.

Generating the crack pattern is a matter of building the web of Points by following simple rules that makes sure the model is being used correctly.

The last step may seem like a simple task, but it is really non-trivial and can be a significant bottle-neck, from a computational point of view, if implemented without caution. We found a solution that is not perfect but works reasonably well and utilizes the benefits of the chosen data structure.

All in all, we are satisfied with the result of our work. Our demonstration software (that is probably not fully optimized) shows that it is possible to simulate interesting glass crack patterns in real time on a standard PC. Even if our solution has some known issues, we think that it fulfils its purpose in approaching the subject from in a different direction than other existing attempts.

// Acknowledgements

We would like to thank Anna Lombardi for providing useful feedback throughout the project. Amir Baranzahi gave us an invaluable lecture on the basics of amorph materials in general and glass in particular. We are also grateful towards Michael Hörnquist for answering some important questions during our work with the loose-piece-identification algorithm and we wish to thank Aida Vitoria for interesting points and discussions regarding the same issue. Last but not least, Tomas Larsson helped us out when we were struggling with the statistical aspects of the model. All of the persons mentioned above work at the Department of Science and Technology (ITN) at Linköping University.

Contents

1 Introduction 1
1.1 Background and Purpose 1
1.2 Sources 1
1.3 Method 1
1.4 Structure 2
2 Simulating the Breaking of Glass in Real Time 2
2.1 The Model 2
2.1.1 Physical Foundation 2
2.1.2 Simplifications 3
2.2 The Implementation 6
2.2.1 Will the Glass Break? 6
2.2.2 Data Structure for Representing Cracks 6
2.2.3 Crack Propagation 7
2.2.4 Loose Piece Identification 8
2.3 Graphical Representation 9
2.4 Known Issues 10
3 The Result 11
4 Discussion 11
References 12
Appendix A: Screenshots from the Simulation 13
Appendix B: System Requirements 15
Appendix C: User Manual 16
Appendix D: Source Code 17

Figures

Figure 1. The Weibull probability function ... 4
Figure 2. Impact stress crack pattern ... 4
Figure 3. Minimal loose pieces ... 8
Figure 4. Finding loose pieces ... 9
Figure 5. The dead-end problem ... 10

1 Introduction

This report aims to present the reader with all relevant information regarding our work with developing a fast method for simulating the breaking of glass. The algorithm and is described in detail, followed by a discussion of the result. The source code of the actual implementation is available as an appendix for the technically interested reader.

1.1 Background and Purpose

Even today, when realistic physics and sophisticated 3D graphics are important parts of many popular computer games, there is no well-known serious attempt to simulate the breaking of glass in real-time. Instead, developers settle with static animations or predefined crack patterns which may appear acceptable in some applications but nevertheless are unrealistic and a sometimes disturbing element in an otherwise appealing game experience.

We wanted to try to develop a simple, fast and reasonably correct physical simulation of crack pattern generation and loose-piece detection in breaking glass. The aim was for it to run in real-time on a standard PC.

1.2 Sources

Even though there has been research done in the area of the fracturing of brittle materials, we have found no publication where the underlying physics have been simplified in such a way that they can be used in real-time applications. Therefore, we have based our algorithm on observations, intuition and a short lecture on the basics of breaking of glass given by Amir Baranzahi at ITN, Linköping University. There are also some references to technical publications and books, as well as online documents.

1.3 Method

Since the purpose of the project was to develop a model imitating a physical process, we began by collecting and discussing known facts about the behavior of cracks in glass materials. In parallel with this, we draw the first outlines for the main parts of the algorithm, as well as some of the more complicated sub-algorithms, such as crack propagation and the method for finding loose glass pieces in the crack pattern.

When we felt sufficiently prepared, we started to program the actual software (done in the C and C++ languages, using OpenGL and GLFW for graphics and user input). During this processing, we paused to test and debug our work regularly.

Notes about decisions made, as well as important e-mail dialogs, physical theory and algorithm outlines were posted to a simple online log that we developed at the start of the project. This simplified the process of writing this report, since all important information was stored in one single document, accessible over the World Wide Web.

1.4 Structure

We have tried to make this report as understandable and logically laid out as possible. We begin by describing the physical facts that our model is based upon, followed by the simplifications and the actual model. Then the implementation of the model is explained in detail, after which the result is presented and evaluated. The source code for the graphical program is attached as an appendix.

2 Simulating the Breaking of Glass in Real Time

The objective was to simulate the breaking of glass in a way that is, to some extent, reasonable from a physical point of view. On one hand we had the aim to make the simulation realistic - on the other we had the restriction on the computations involved - the simulation was to run in real-time. This required a model quite simplified compared to the reality and a well-planned implementation. The following section will present our thoughts and decisions made during the development process.

2.1 The Model

Glass is a very complex material. There is no exact formulas which can tell how glass will shatter, and the formulas there are demand a high level of knowledge of physics and chemistry. We didn't have the time to learn all the things we would need to understand everything, and we needed a simplified model for the simulation to run in real-time. We decided to go through the physics to get an overview, then extract the properties that have the largest impact on how shattered glass really looks, and to build a model based on this.

2.1.1 Physical Foundation

The following text is based on a conversation with Amir Baranzahi (ITN, Linköping University) that took place on 10 November 2004, when not stated otherwise.

Glass (SiO_2) is an amorphous material, which means that the material is a mix between a solid and a fluid. On a microscopic level, this means that in glass the atoms don't lie in straight lines but instead a bit random. Due to this it's almost impossible to say exactly how a crack will propagate in the material - you never know on which side of an atom a crack will go. What you can say is that the crack likely will continue in the same direction as it started in.

Whether the glass will crack at all is dependent on how hard the impact is. This can differ a lot from object to object, even if they are of the same size, shape and have the same frames. There is a statistical distribution though, the Weibull distribution, that can give a hint as to what the probability of cracks forming would be for a given input stress (Askeland, 1996). Glass have a Weibull number of 5-8 (Lower, exact date unknown) (see figure 1), with low numbers implying a wider distribution (steel has a Weibull number of about 40).

A window will stand a harder impact in the middle of the glass than at the edges. This is because at the edges the glass is fixed to a frame and therefore it cannot oscillate as freely as in the center, thus absorbing less stress and breaking more easily.

Inside the glass there are tiny air bubbles that formed when the glass solidified during the creation process. When a crack comes to an air bubble, its behavior is undefined; it can change direction completely,

divide itself into several new cracks or perhaps stop completely. This is because the crack will continue in the direction where the "edges" in the air bubble are the weakest, and there may be several such spots. The number of air bubbles in a glass is impossible to tell in advance, but if there are visible ones, there are probably many more that are so small that you cannot see them.

A crack propagates with a very high speed. This differs from different sorts of glass, but in general you can say that the speed is approximately 1950 m/s (Weeks, 2004), compared to a bullet that travels in about 800 m/s. This means that you will never be able to see a crack spread in a window glass.

When a strike hits a window, cracks will form if the glass bends more than it can withstand. Glass as a material is quite non-elastic, which is why a window easily cracks . If a strike is hard enough, circumferential crack patterns will form (see figure 2). Also, the number of radial cracks will be larger (Mencik, 2002). There is no way of telling how many cracks will form when the window is hit by a given blow, but in general you can say that if you hit harder, more cracks will form. There will almost always be cracks forming at opposite directions from each other, i.e. if one crack propagates out from the impact point at one direction there will be another that starts off with the angle of the other one plus 180 degrees.

When you hit hard enough the holes in the window will no longer get larger, but smaller. This is because the object you hit with will go through the glass before it has time to expose the material to any significant amount of bending stress. This also means that the cracks that form become shorter (Mencik, 2002). That is why there can be small holes of sizes almost equal to the bullets' when certain types of glass windows have been shot upon.

A glass that already has cracks or holes or other flaws in it will break on a much lighter impact. The cracks will behave and propagate in an even less predictable way (Askeland, 1996).

2.1.2 Simplifications

As mentioned above, to really understand how glass shatters, you need a lot of special knowledge. If you want to implement this knowledge with a precise physical model you will have to do a lot of processing before you will see the result. This kind of realistic simulation of objects cracking has been done before, and included heavy pre-animation computations (Hiroto et al., 1998). One of he things that we wanted to achieve with this project was a simulation that should run in real-time. This implied that we had to do a lot of simplifications while still preserving a graphically plausible result. The simplifications made are based on the physics presented above, as well as our own experience with breaking glass and crack patterns.

The first and perhaps most important simplification that we made was to restrict our simulation to a two-dimensional surface. This means that our model only takes into account cracks that form in the plane of the window pane, which in turn is assumed to be "thin". Also, since many of the characteristics of crack patterns depend heavily on the type of glass, we have limited ourselves to considering the type of window glass that can be found in a typical display window of a shop. That is, not exactly bullet-proof but not really wine-glass-thin either.

Initially we wanted to be able to hit the same window several times, but we soon decided just to allow one hit. Because of this we could set up a simpler and more general model for how the cracks will propagate.

A simplification that is perhaps not so easy to justify is our choice of "impact force". Since we could not find any facts about the relationship between the breaking point of the glass and the velocity, size or mass of the "crushing" object, we decided to assume that the object is of a fixed size (about the size of a tennis ball in this case), has a constant (unknown) mass and hits the glass pane with some impact velocity. The

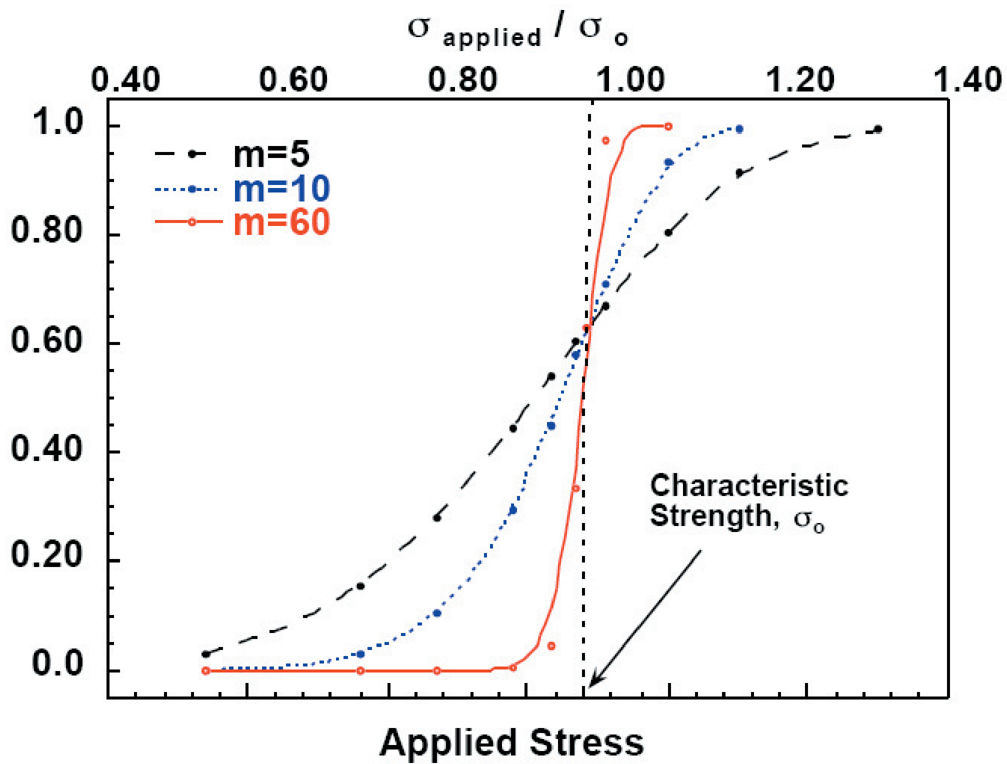


Figure 1. The Weibull probability function

Glass has a Weibull number of between fire and eight. In the figure, 'm' is the Weibull number. The higher the number, the more determined is the breaking point of the material. Having a relatively low number, glass will exhibit a breaking behavior that is quite hard to predict.

Figure 2. Impact stress crack pattern

The photo shows a cracked glass window pane where circumferential cracks have formed due to impact stress.

important thing is that the breaking point should be able to vary according to the Weibull distribution and also become lower if you hit closer to the frame of the window. We decided to let 10 m/s be the velocity that would cause the glass to break if hit in the center, on average. At a velocity of 35 m/s (on average) and above there would no longer be any cracks around the hole caused by bending-stress. We set a maximum at 40 m/s just to be sure that about every possible instance of the glass window will have all its possible breaking patterns included. The number does not really mean anything and has no real physical correctness - it is just needed for the model to be able to show the different types of crack patterns that can arise from the breaking of a glass pane. We could probably easily adjust this to be more correct if we had any facts to base this particular part of the simulation on.

The number of cracks will depend of how hard you hit the glass and the angles will be randomized so that they can propagate at almost any directions. We have two rules when randomizing the angles. The first rule is that the difference in angle between two neighboring radial cracks will not be below 12 degrees. This number is dependent of the maximum number of radial cracks originating from the point of impact, which we set to 30. More cracks (or cracks closer to each other) will not contribute in a positive way to the visible pattern, but just make the computations more cumbersome. The second rule is that all radial cracks will always have another crack that is directly opposite in direction to the first one. For example, if the simulation has decided that there should be a radial crack pointing out from the impact point at an angle of 30 degrees, this implies that there will be a second crack at an angle of 30 + 180 = 210 degrees.

We also decided to make the radial cracks that are connected to the circumferential cracks completely straight. That is not completely realistic, but we thought it reasonable. The main reason for this simplification (as for many others) was to reduce the calculations needed. We did an assumption that the first circumferential crack will always form at a distance from the impact point that depends on how hard the impact was and after that the distances between subsequential circumferential cracks would get shorter and shorter before they would stop forming completely. Also this seemed a reasonable assumption to make. If you hit with an impact velocity just over the breaking point there will be no circumferential cracks at all. The model does not create the circumferential cracks in the same distance on every radial crack; the distances can differ a bit. This makes it look more like real crack patterns.

After the forming of the circumferential cracks has ceased, the radial cracks are able to change direction slightly as they propagate, just as they can in reality. Each radial crack is assigned some kind of “energy”, in terms of how long it can get at a maximum.

We also model the probability and effect of cracks hitting air bubbles. Cracks do not divide into several new ones if the crack that hit the air bubble is close to run out of energy. This represent our assumption that a certain amount of energy is needed for the outgoing cracks to break through the “walls” of the air bubble.

We use a linear model when calculating the number of circumferential cracks, their distances to the impact point and so on. This is probably not physically correct - in fact, we have not found any written material that states any facts about how it really works - but it looks good enough and makes computations fast and simple.

2.2 The Implementation

We broke down the problem into four separate, logical steps, namely deciding whether and how the glass breaks, generating the crack pattern from these starting conditions, building of a data structure to represent the pattern and finally identification of the loose pieces that may or may not have formed in the pattern. In the following sections, each of these steps are described more in detail.

2.2.1 Will the Glass Break?

The first thing that happens when you have hit the glass with a certain speed and at a certain place on the window glass is that the program calculates if the glass will break at all. This is first done with a random function that count the break limit of this glass according to the Weibull distribution (see Physical Foundation above). After that, the program includes the spot where the glass was hit in the calculation and decide if the glass will break (see Simplifications above). If the glass did not break, no further calculations will be carried out. Otherwise, the program continues with calculations of how many cracks that will form, together with their directions (see Simplifications above). Depending on the magnitude of the impact velocity the program will then decide if any circumferential cracks should form. If that is the case, their distance to the impact point are calculated. This is done a bit random - the distance is decided and then a random number is added to it.

2.2.2 Data Structure for Representing Cracks

To represent the generated cracks, some kind of efficient data structure is required. When the simulation is to be run in real time, it is important that this structure can provide fast access to its data in all steps of the algorithm - from collision detection (line-line-intersection check) to traversal for drawing purposes.

First of all, we needed a representation for the actual cracks. It was decided that a crack is made out of a number of segments, where each segment consists of two connected points in the plane of the glass window pane. This representation was chosen because of its simplicity, ease of implementation and flexibility - it is easy to change the crack resolution (and computational burden) by adjusting the crack segment length (and hence the total number of points).

Second, we had to decide how the data structure was to be implemented. Since we defined cracks as a number of connected points, the logical choice was to create a Point class and for each crack instantiate a number of Point objects. Each Point contains information about its own coordinates together with a list of all connected Points and the respective angles of these connections. The Point also has a couple of "flags" used in crack propagation, loose piece identification and drawing functions.

The Point objects have to be stored somewhere where they are easily accessible from all aspects of the algorithm. We decided to use a Vector for this purpose. A C++ Vector provides access to a specified position (Point) and appending of new Points in constant time. Searching for a Point given certain properties or inserting a Point at another position than the end of the Vector takes linear time. All in all, it is a flexible list that fit our purposes.

To summarize; the data representation of the crack pattern in our solution is a Vector of Points, where the connections between the Points - defining the actual cracks - are stored in the Points themselves. Angles between the connections are calculated one time only and stored in the Points in the same way. These angles are used both for crack propagation and loose piece identification later on in the process.

2.2.3 Crack Propagation

Once all the starting conditions for the crack pattern have been calculated and the initial data representation has been initialized, the algorithm proceeds with the actual generation of cracks. This is done recursively, one Point at a time, which is why we chose to refer to it as "crack propagation". The propagation consists of two individual phases: the first creates cracks due to impact stress and the second creates cracks caused by bending stress. For a brief description of these phenomena, see the section "Physical foundation" above.

The impact stress phase consist of positioning a Point at the impact coordinates (which are read from mouse input, in our application), then drawing radial cracks, with length determined by how much energy each of them has. On these radial cracks, at predefined distances to the impact point, near-perpendicular cracks are created, joining with the closest other radial crack to the "right" (clockwise orientation) to form the spider-web-like pattern. At the end of each radial crack in the spider-web, bending stress crack propagation is initiated, if there is any bending energy in the crack (calculated previously, depending on the impact velocity).

The bending stress propagation is represented slightly differently depending on, if the impact velocity was large enough to obtain any impact stress or not. In the case where there are no impact stress the first points of the crack is positioned out randomly around the impact point, the only condition is that the angle between two of the starting crack segments is greater than twelve degrees, a limitation set for viewing purpose. From here on the continuing process is equivalent, no matter if there is impact stress or not. In the last created points an angle is calculated, which may deviate from the "incoming" angle to this Point by only a little. Then a new Point is created a certain step-size away from the current Point, in the direction of the calculated angle. Depending on the size of the step, the bending energy is decremented. Collision detection is preformed to make sure that the crack does not cross any other crack in the pattern. If so, the cracks are joined, with appropriate adjustments to the data structure, which may result in loose pieces, making the pattern more realistic and interesting. The mentioned collision detection is really a quite simple line-line intersection test, where the proposed new segment is tested against all existing segments. This is a slow but straightforward approach.

In every step there is a slight possibility that the crack has entered an air bubble. This is represented with a random function which may split the crack into maximum two new cracks propagating away independently of each other. Since the structure of an air bubble is impossible to predict the "new" cracks may propagate in any direction, except for the incoming direction, which is also a limitation for viewing purpose. If the bending energy left in the crack is very small, the crack will not split in an air bubble. When all the energy is used, the crack stops.

After the crack pattern resulting from both impact and bending stress has been generated and stored in the Vector of Points, the crack propagation step of the algorithm is finished.

Mainly due to the naive intersection test used, the crack propagation takes quadratic time, which could certainly be improved for faster execution.

2.2.4 Loose Piece Identification

One of the main parts in our application was to come up with an algorithm to find all the holes in the window created by the crack pattern. The idea is to find the "smallest" holes (the holes corresponding to the smallest loose glass pieces), and no holes that consists of other holes. An example of this is shown in figure 3. This criteria was set because we wanted to make it possible to extend the application further with pieces falling off the glass window.

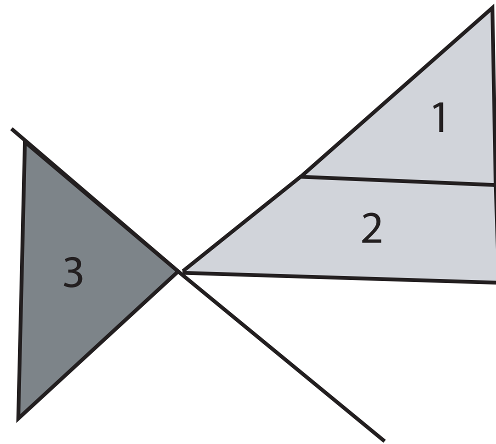


Figure 3. Minimal loose pieces
The figure shows three different holes, all numbered independently. Hole number one and two is colored in the same way since they together form a large hole. The goal of the loose piece identification is to find each separate piece, even if they together could count as a single hole.

To start the process of finding all the holes, the first thing to do is to eliminate all the cracks that are not important, which in this case are the "dead ends" if the pattern is imagined as a maze. When all the dead ends are removed from consideration, the method is as follows:

1. Start at an arbitrary Point in the pattern. We start the trace for holes from the impact point, but the algorithm is not depending on this.

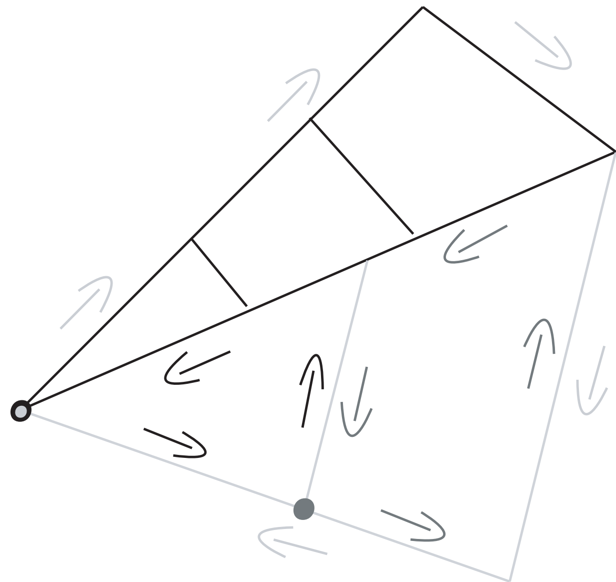
2. From the current point, go to the next by always choosing the left most connection.

3. If arriving back at the starting point we have a closed loop of connected Points describing the edge of a hole, and a check is needed to find out whether it has been traversed clockwise or counter-clockwise. If the direction was counter-clockwise, the hole is guaranteed to be minimal in the way we are interested in. If the orientation is found to be clockwise, there is no such guarantee, which in turn implies that a larger hole (containing several minimal holes) is found. This is easy to understand with a figure, but how to implement it is perhaps not as clear. Our solution is to check all Points in the loop and their connections to other Points. For every new Point (which corresponds to a corner in the polygon defined by the loop) we calculate the angle between "incoming" and "outgoing" segments, that is, the left most connection viewed from the incoming one. This is a simple operation since the angles are saved in every Point at the creation of the crack pattern. At arrival at the starting point, if the sum of all the angles are 180 * (n - 2), where n is the number of corners in the hole polygon, the polygon was traversed counter-clockwise, otherwise the sum will be 180 * (n + 2) and traversing was clockwise.

Now we know how to find all the "smallest" holes, but there is still a possibility to find the same hole several times. Therefore we added a condition that a segment may only be traversed once in each direction. A small example to make it a bit clearer is shown in figure 4.

Thanks to the design, the loose piece identification is carried out with linear time complexity.

Figure 4. Finding loose pieces
The black dot is the starting point. Black arrows indicate the first hole found, taking the left most connection at every point. At returning to the start point it is checked that the traversal was done counter-clockwise. Starting again in the same point now marked as light gray (almost white), the next path is indicated by light gray arrows (always using the left most connection). When finished, the sum of the angles indicate a clockwise traversal, which means that the hole is not stored since there is no guarantee that it is minimal. Since all the connections from the start point have now been used, the next point, marked with a dark gray dot, is the new starting point. Traversing again gives the path of the dark gray arrows and a hole is found (since counter-clockwise orientation). Now, all the connections from the dark gray point are used once in each direction, so the dark gray point is regarded as finished and the next point is chosen as the new starting point. The figure shows the example at this point. All connections that have been used in both directions are colored light gray (almost white).



2.3 Graphical Representation

To visualize the simulation we used OpenGL to create a wall containing a window which in turn is a transparent glass pane. The glass pane is made out of only two triangles and is thus in itself only defined in two dimensions (although it is truly positioned in a 3D-environment). We used alpha blending on the glass pane which is a technique that simulates transparency. In addition to the three color channels for every pixel in the pane of glass, a fourth transparency channel, alpha, is used. This alpha channel is used by the frame buffer to blend the pane with the environment behind the glass.

Since the simulation is executed in two parts we decided to separate the visualization in two steps. In the first part we draw the lines that represent the crack pattern. The second part is to visualize any holes that might have formed in the crack pattern. The two parts is executed separately, each triggered by input from the user.

In the first part the lines representing the crack pattern is created by iterating through the Vector containing all Points, and for every Point lines are drawn to connected Points. A boolean “flag” is used to indicate if a connection already has been drawn. The lines are drawn independent of the glass pane but in the same plane. This makes the lines to look as if they where a crack pattern in the glass pane.

The second part of the visualization is more complex then the first one. OpenGL is designed for low-level operations and can only render simple convex polygons - that is, for every line that connects to points on the boundary of the polygon, the line is contained entirely inside the polygon. Our glass containing holes must therefore be subdivided into simple convex polygons before they can be rendered. This subdivision is called “tessellation”, and the OpenGL Utility Library (GLU) provides functionality for doing the actual work. The tessellation function requires that the coordinates of the points is in the same order as they appear in the path that make out the hole. Since we automatically store the holes in the correct order when we use our hole detection algorithm, this was straightforward to implement. The tessellated object is stored so that the tessellation only needs to be performed once and can later be reused in subsequent frames.

Since the drawing of the crack patterns is independent of the tessellation of the cracked window, we had to modify the drawing so that only the cracks not being a part of any hole are drawn while the loose pieces

are still in the window. The partial solution we came up with was to draw only the cracks that are "dead ends" in the pattern. A crack segment is considered part of a dead end if it has less than two non-dead end connections. A boolean flag indicating this had already been set during the algorithm that checks if a crack is a dead end. Using this method gave rise to a small issue, explained in detail in the next section.

2.4 Known Issues

One of our main goals was to make our algorithms as general as possible, expecting that this would result in few issues. Despite this, two issues not to be overlooked rose.

The first problem occurs when the velocity of the colliding object is high enough to contain impact stress energy. When the resulting radial cracks or circumferential cracks are calculated and if some crack is given an angle that is an integer multiple of 90 degrees, this crack doesn't connect to other cracks in a proper way - the collision detection methods somehow cannot handle it. To get around this we decided to increase such angles by 0.1 degrees. This will not significantly alter the visual result but eliminates the bug.

The second problem is a result of simplifications in our implementation. In the second step of our visualization when the created holes are shown, we only draw the cracks that are dead-ends, since other cracks are part of loose pieces and are thus removed. In almost every case this gives us our expected result, but when two not-adjacent holes are connected corner-to-corner by a crack (see figure 5), it does not. The crack joining the two holes is by the algorithm that finds dead-ends (correctly) not marked as one and will therefore not be drawn in the second visualization step. The same problem also occurs if there is a dead-end inside a loose piece. After the removal of the piece, the crack is still drawn, appearing to be hanging in the air. If we give the tennis ball a velocity that maximizes the chance of getting this wrong result, these bugs occur approximately 1 out of 30 times. This could be fixed by implementing the loose piece identification algorithm in a smarter way.

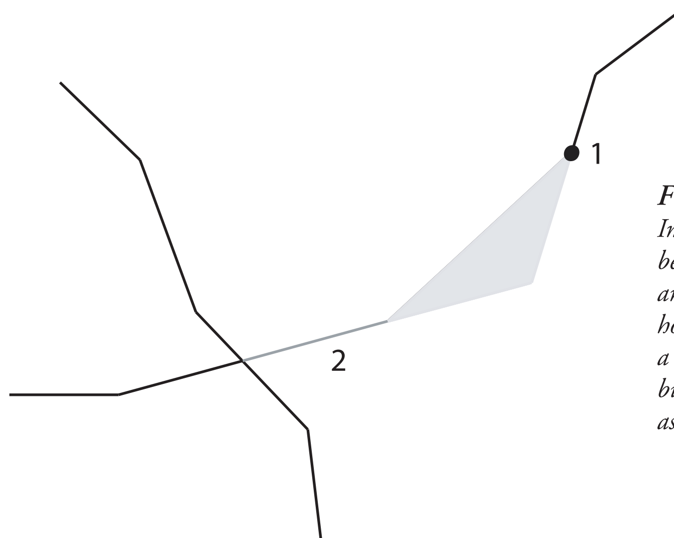


Figure 5. The dead-end problem
In this figure, the light line (marked as 2) will not be drawn, since it is not registered as a dead-end and the dead-ends are the only lines drawn after the holes are found. This type of hole occur only when a crack splits because of an air bubbule. The air bubble is indicated with a large black dot (marked as 1) in the figure.

3 The Result

The first step in our simulation, whether or not the glass pane cracks, is implemented in a more or less physically correct way, even though we make use of simplifications. Since we use a randomization function based on a discreet Weibull distribution, this is also fast and could easily be adjusted to a different resolution.

Concerning the accuracy of the model, we can see in the figures of appendix A that the generated crack patterns compares well to the real glass fracture in figure 2 of the report. Our model has, despite the simplifications, shown that it gives acceptable to good results for most impact velocities. It is hard for us to evaluate the results for extremely high velocity impacts since this is a rare phenomena in the real world. Screenshots from the simulations taken at different velocities can be found in appendix A. Even though we use a simple linear model for the distribution of energy between bending stress and impact stress, the results are good. With a higher knowledge of the complex physics involved, the model could quite easily be altered to simulate the reality in a better way.

A lot of effort was put into the algorithm that finds the loose pieces. It has been tested numerous times with complex crack patterns and it works good, as expected. The data structure we came up with largely made this possible.

In the loose piece identification algorithm, all the possible glass pieces are marked as if they will fall out of the glass pane. This is not always the case in a real situation, where some pieces may still be in place even though they are surrounded by cracks. Our model does not take this into account, but since the pieces are identified and stored, an extension of the program could model this behavior as well as pieces falling off the pane.

One of the most important goals was to make the whole simulation run in real time, which succeeded in the sense that a high-end PC with a decent graphics card can do it. For recommended system requirements, see appendix B. When the velocity is high, resulting in many cracks, there is a delay for a fraction of a second. This is noticeable when the camera is moving at the same time as the crack pattern is calculated and visualized. The most time consuming part is the (naive) collision detection that checks if a crack intersects another crack, which we believe to be the "bottle neck" of the crack generation algorithm.

To summarize, the final program has some flaws but still produces realistic and interesting crack patterns. The loose pieces are always found, something that is done in linear time. It is fair to say that we managed to reach the goals we had set up for this project.

4 Discussion

Even though we are satisfied with the result of our work, there are a number of points that could be improved. First of all, there is the obvious business of animated loose pieces falling off the window pane. This we skipped since it was not really connected with the main issue of producing realistic crack patterns, but we did prepare for such an extension by being thorough in our work with the loose piece identification. Not so thorough, though, that the solution is flawless - there is currently the problem with some "dead ends", indeed uncommon, but still there. This could probably be solved in some more or less simple manner, but we have failed to produce such a solution.

The method could of course be considered improved if we had added support for multiple strikes. This was our intention from the beginning, but we decided to concentrate our efforts on other factors after we

realized how complex the problem would get. What if the user with the second strike hit a piece of glass that hung to the frame by a single, tiny shred? Intuition would call for the entire piece to fall off with the shred cracked, if the strike was moderate. If it was severe, perhaps only part of the piece would break? We found no good way to simplify this complex matter, and we would rather skip it than produce an ugly solution. Hence the limitation to one strike per window pane, as described earlier.

The collision detection method used during the creation of the crack pattern is quite naive and slow when a small stepsize is used - the current crack segment is checked for intersection with every other existing segment. This could be improved, for example by dividing the pane into several smaller regions and handling intersection tests separately within this regions as far as possible. This would probably speed up the process, but with this project we wanted to try the crack creation and loose piece identification methods, not develop a sophisticated collision handling system. This is why we went for the obvious solution.

On the good side, we can conclude that our method produces quite realistic crack patterns in very little time, which is what we aimed for. We think that we have succeeded in simplifying a very complex problem into something a standard PC can handle in real time, while still preserving many of the visual characteristics of the real-world counterpart. We think that our method produces crack patterns that looks better than any other real time attempt that we have seen, including current computer games like Half Life 2. Our method for finding loose pieces is very fast and could probably be used in a real application.

If the method would be used in a computer game or some such, some modifications would probably have to take place. Even if the simulation can run in real time on its own, there are many more things for the computer to handle when the glass-breaking is not the main activity. The step-size could be reduced, the collision detection method replaced and the code optimized in general. Still we believe that the idea behind our model could be used to improve today's real time simulation of the breaking of glass.

References

Mencik, Jaroslav (2002). *Strength and fracture of glass and ceramics*. Elsevier.

Baranzahi, Amir of ITN at Linköping University. Personal conversation (2004-11-10).

Askeland, Donald R (1996). *The Science and Engineering of Materials*. Thomson Learning. 3rd edition.

Hiroto, Tanoue, Kaneko (1998). Generation of crack patterns with a physical model.

Lower, Nathan. *Weibull Statistics* [www]
<http://web.umr.edu/~nplc4b/Weibull.html> (Verified 2004-12-18)

Weeks, Joseph (2004). *How fast does a crack in a glass travel?* [www]
<http://www.madsci.org/posts/archives/2004-07/1090933065.Ph.r.html> (Verified 2004-12-18)

Appendix A: Screenshots from the Simulation

Below are eight screenshots from the actual simulation, demonstrating the generated crack patterns for different impact velocities. The images have been processed for contrast enhancement.

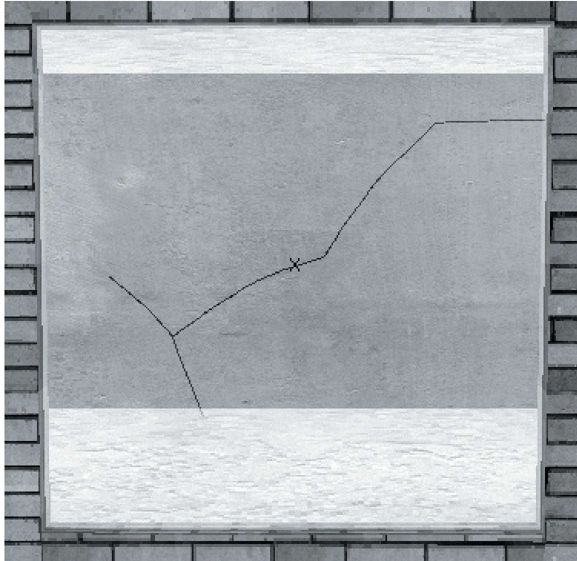


Figure A-1. Impact velocity 10

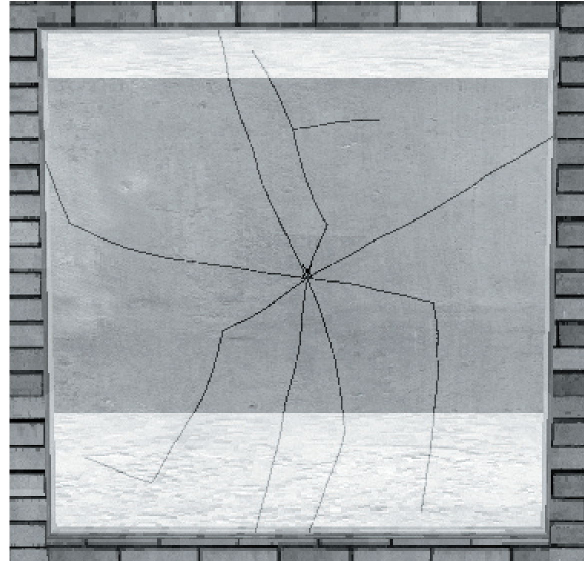


Figure A-2. Impact velocity 13

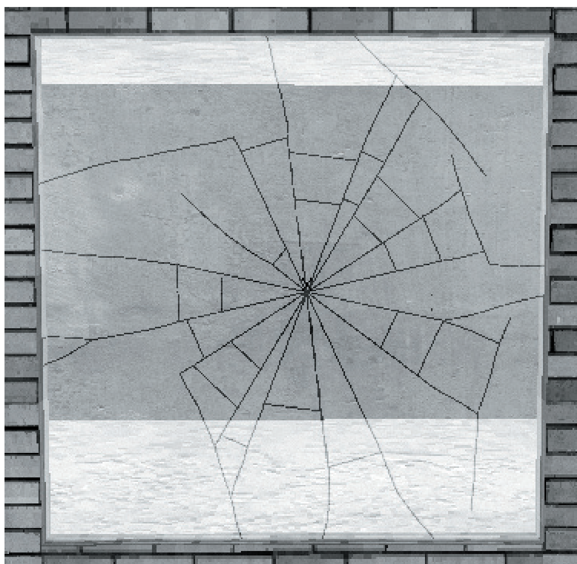


Figure A-3. Impact velocity 16

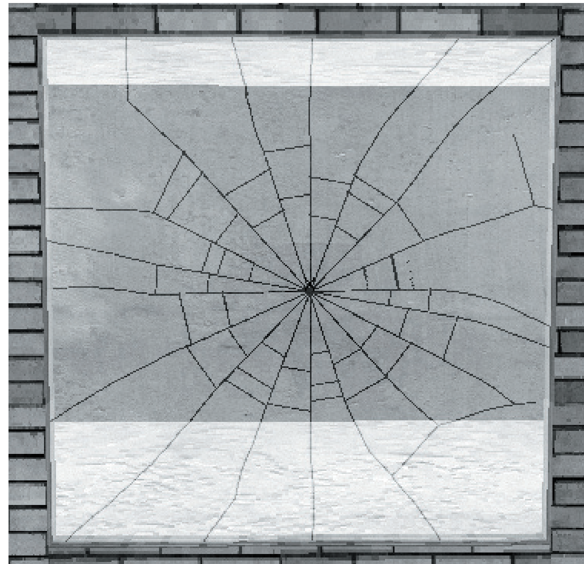


Figure A-4. Impact velocity 19

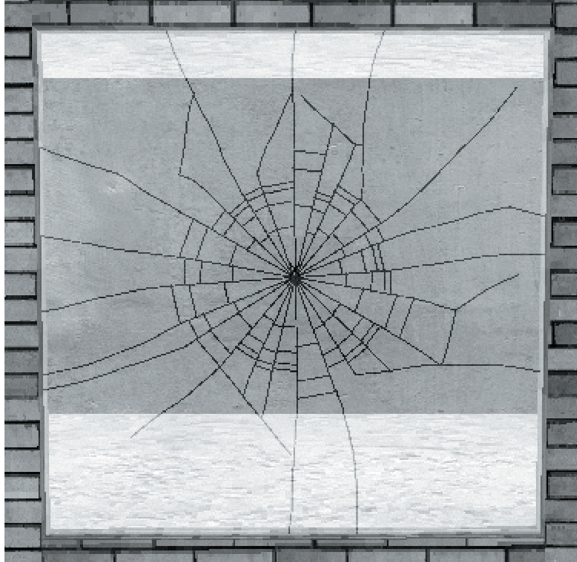


Figure A-5. Impact velocity 22

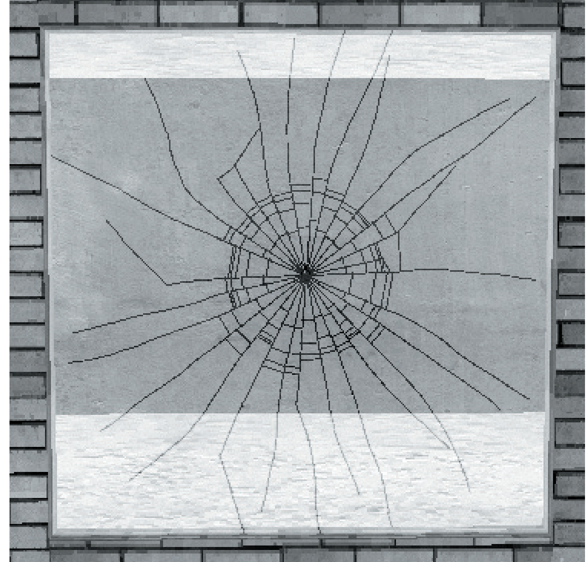


Figure A-6. Impact velocity 25

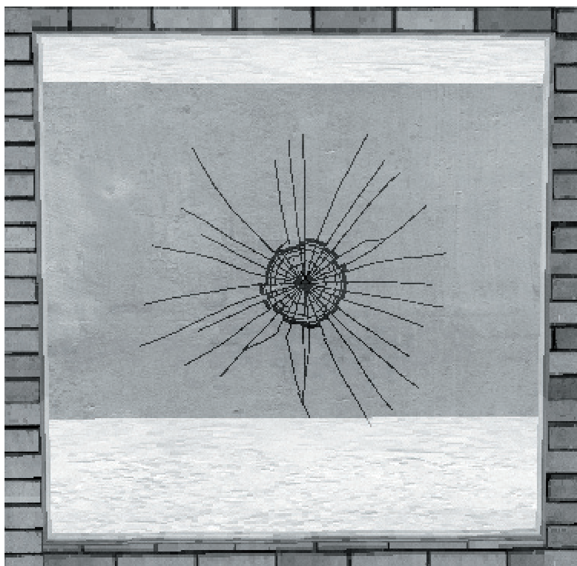


Figure A-7. Impact velocity 30

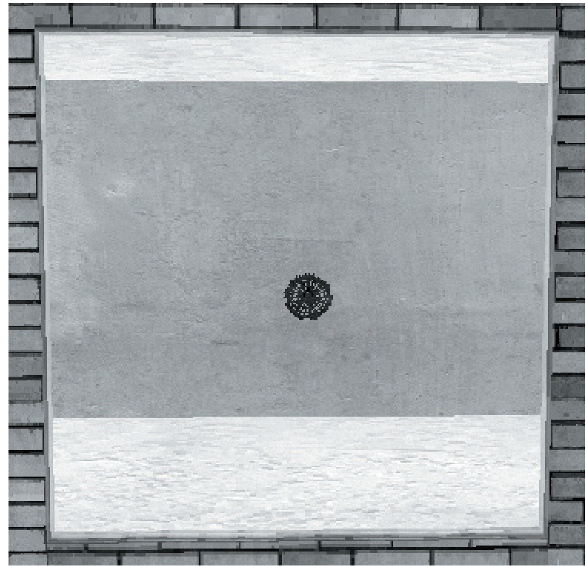


Figure A-8. Impact velocity 40

Appendix B: System Requirements

The simulation has not been tested on that many machines, but we have found it working excellently on a PC with the following hardware:

AMD 64 3500+ or equivalent
32 Mb of working memory (RAM)
Nvidia GeForce 6800GT or equivalent
7 Mb of free harddrive space
Microsoft Windows with OpenGL installed

The program will probably run quite smoothly even if the machine is not as powerful as the one above, but lesser "lags" could occur. Generally, an OpenGL-enabled graphics card is highly recommended. The source code should compile under other operating systems as well, possibly with some modifications needed. For this, OpenGL, GLU and GLFW are all required.

Appendix C: User Manual

We have tried to make our program quite easy to use, even though we must admit that the user interface has not been a top priority issue.

The simulation runs in full-screen mode, but the resolution can be changed to match the available hardware. This is done by supplying a couple of extra parameters at the command line. The following command starts the program and sets the resolution to 1600 x 1200 pixels:

```
splitter.exe 1600 1200
```

If no resolution parameters are specified, the resolution will default to 1024 x 768 pixels.

Once the program is running, use the mouse to look around in the virtual environment. It is possible to move around and control the simulation using the following keys:

W or UP-arrow	Move forward
S or DOWN-arrow	Move backward
A or LEFT-arrow	Move left
D or RIGHT-arrow	Move right
Keypad +	Increase impact velocity
Keypad -	Decrease impact velocity
Left mouse button	Hit the glass window where the crosshair points
Right mouse button	Remove loose glass pieces after a crack pattern has formed
SPACE	Reset the simulation
ESCAPE	Quit the program

Appendix D: Source Code

```
/* splitter.cpp - The main application file of the Splitter project. This file
 *                contains the main program loop.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <windows.h>
#include <GL/glfw.h>
#include <cstdlib>
#include "graphics.h"

using namespace std;
using namespace splitter;

/* Main method, initializes everything, includs the main loop
 * that ends when Esc-key is pressed
 */

int main(int argc, char** argv) {
    bool running = true;

    if (argc >= 2) {
        initGraphics(atoi(argv[1]), atoi(argv[2]));
    } else {
        initGraphics(1024, 768);
    }

    initUIInput();
    srand(time(0));

    while (running) {
        showFPS();
        draw();

        //Swap the drawing buffers
        glfwSwapBuffers();

        //Check if esc is pressed or if the window is closed
        running = !glfwGetKey(GLFW_KEY_ESC) &&
                        glfwGetWindowParam(GLFW_OPENED);

    }

    deinitGraphics();

    return 0;
}
```

```
/* crackCreation.h - Contains functions for creating the crack pattern,
 *                  crack propagation with collision detection between cracks.
 *
 * Authors:
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include "Point.h"
#include "CrackStarter.h"

using namespace std;

namespace splitter {

    /* Used by safeConnect(). This function checks whether the supplied points
     * form lines that intersect closer to the start point than the last found
     * intersection.
     *
     * Input:  (point1) First point used in the first line.
     *         (point2) Second point used in the first line.
     *         (thisPoint) The start point of the proposed new line.
     *         (newPoint) The end point of the proposed new line.
     *         (oldDist) The city-block distance from thisPoint to the nearest
     *         previously detected intersection. Should be 0 to indicate that no
     *         intersection has been detected before.
     * Output: TRUE if a new intersection is found (closer than any previous
     *         one),
     *         FALSE if no intersection or intersection but further away than a
     *         previous one.
     */
    bool findNewIntersection(Point* point1,
                             Point* point2,
                             Point* thisPoint,
                             Point* newPoint,
                             float* oldDist);

    /* Connects a point to another one. If the connection between these points
     * is crossing previously made connections, appropriate actions are taken.
     *
     * Input:  (thisPoint) the point in question, where a connection is to
     *         occur.
     *         (newPoint) the second proposed point for the connection wanted.
     *         (allPoint) a vector including all existing points and their
     *         connections.
     * Output: (bool) True if there is a collision between the expected
     *        connection and any other existing connection. False if no
     *        collision accured.
     *        thisPoint will be connected to another point in any case.
     */
    bool safeConnect(Point* thisPoint, Point* newPoint,
```

```
                    vector<Point*>* allPoints);


    /* Propagates a crack until its energy is zero or the crack collides with
     * another already existing crack.
     *
     * Input:  (thispoint) point from where the crack segment are to be
     *         generated
     *         (prevAngle) the global angle on the previous crack segment
     *         (bendEnergy) how much bending energy left in the crack
     *         (allPoints) vector that includes all the different points and
     *         the internal connections
     *         (deadEnds) vector that includes all the end points in the
     *         crackpattern and their connections
     * Output: void
     */
    void propagate(Point* thisPoint, float prevAngle, float bendEnergy,
                  vector<Point*>* allPoints,vector<Point*>* deadEnds);

    /* Creates the data representation of a crack, from impact to stationary
     * phase.
     *
     * Input:  (starter) CrackStarter object with init info for the propagation.
     *         (allPoints) Vector containing all the crack segment points.
     *         (deadEnds) Vector that includes all the end points in the
     *                   crackpattern and their connections.
     * Output: None.
     */
    void createCrack(CrackStarter* starter,
                    vector<Point*>* allPoints, vector<Point*>* deadEnds );

}
```

```
/* crackCreation.h - Contains functions for creating the crack pattern,
*                   crack propagation with collision detection between cracks.
 *
 * Authors:
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <iostream>
#include <math.h>
#include <cstdlib>
#include "Point.h"
#include "CrackStarter.h"
#include "crackCreation.h"
#include "holes.h"

using namespace std;

#ifndef SPLITTER_PI
#define SPLITTER_PI
const float PI = 3.14159265;
#endif

namespace splitter {

    const float STEPSIZE = 0.2f;
    const float ENERGY_LOSS = 0.025f;
    const int MAX_BUBBLE_CRACKS = 3;     // Not allowed to be greater than 30
    const int MIN_DIFFERENCE_ANGLE = 12; // Guarantees visibility of cracks

    bool findNewIntersection(Point* point1, Point* point2, Point* thisPoint,
                             Point* newPoint, float* oldDist) {

        float x1 = point1->getX();
        float y1 = point1->getY();
        float x2 = point2->getX();;
        float y2 = point2->getY();;
        float x3 = thisPoint->getX();
        float y3 = thisPoint->getY();
        float x4 = newPoint->getX();
        float y4 = newPoint->getY();

        // Calculate intersection coordinates. If den is 0, there is no
        // intersection between the two lines. If den is not 0, there is an
        // intersection at apoint along the (infinitely long) lines, with
        // coordinates (xnum/den, ynum/den). That point may, however,
        // lie outside of thesupplied points, in which case the intersection
        // is not interesting.
        float den = x1*y3-x1*y4-x2*y3+x2*y4-x3*y1+x4*y1+x3*y2-x4*y2;
        float xnum = x1*x3*y2-x1*x4*y2-x2*x3*y1+x2*x4*y1-x1*x3*y4+
                     x1*x4*y3+x2*x3*y4-x2*x4*y3;
```

```
    float ynum = x1*y2*y3-x1*y2*y4-x2*y1*y3+x2*y1*y4-x3*y1*y4+
                 x4*y1*y3+x3*y2*y4-x4*y2*y3;



    bool intersection = true;

    if (den == 0) {
        intersection = false;
    } else {
        float x = xnum/den;
        float y = ynum/den;

        // Check if intersection lies between the given points on the lines
        if (((x > x1 && x > x2) ||
             (x < x1 && x < x2) ||
             (x > x3 && x > x4) ||
             (x < x3 && x < x4)) ||
               ((y > y1 && y > y2) ||
             (y < y1 && y < y2) ||
             (y > y3 && y > y4) ||
             (y < y3 && y < y4))) {

            intersection = false;
        } else {

            // Check if this intersection lies closer to the start point
            // of the proposed new line than a previously found one.
            // If so, this is the intersection to remember. If not, it is
            // discarded.
            float testDist = abs(x3 - x) + abs(y3 - y);
            if (testDist < *oldDist || *oldDist == 0) {
                newPoint->setX(x);
                newPoint->setY(y);
                *oldDist = testDist;

            } else {

                intersection = false;
            }
        }
    }
    return intersection;
}



bool safeConnect(Point* thisPoint, Point* newPoint,
                 vector<Point*>* allPoints) {

    Point *p1 = NULL;
    Point *p2 = NULL;
    float tempDist = 0;
    bool collision = false;

    //Marks the connections belonging to the point being checked
    for (int i = 0; i < thisPoint->getNumConnections(); i++) {
        thisPoint->setTestFlag(i, true);
    }

    //Checks for collisions with all other connections
    for(int i = 0; i < allPoints->size(); i++) {
        if(allPoints->at(i) != thisPoint) {
            for (int j = 0;
                 j < allPoints->at(i)->getNumConnections();
                 j++) {

                if(allPoints->at(i)->getTestFlag(j) == false) {
                    if (findNewIntersection(allPoints->at(i),
                                    allPoints->at(i)->getPoint(j),
                                    thisPoint,
                                    newPoint,
                                    &tempDist)) {

                        p1 = allPoints->at(i);
```

```
                        p2 = allPoints->at(i)->getPoint(j);
                        collision = true;
                    }
                }
                //Resets the flags for all the connections when arriving to
                //the second point in the connection
                allPoints->at(i)->setTestFlag(j,
                                    !(allPoints->at(i)->getTestFlag(j)));

            }
        }
    }

    //In case of a collision connect the new connection with the one
    //colliding with and modify the previous connection. If collision with
    //an exact point, connection is made to the point in question.
    if (collision) {
        if (newPoint->getX() == p1->getX() &&
              newPoint->getY() == p1->getY()) {

            thisPoint->connect(p1);
            delete newPoint;

        } else if (newPoint->getX() == p2->getX() &&
                 newPoint->getY() == p2->getY()) {

            thisPoint->connect(p2);
            delete newPoint;

        } else {
            thisPoint->connect(newPoint);
            p1->disconnect(p2);
            p1->connect(newPoint);
            p2->connect(newPoint);
            allPoints->push_back(newPoint);
        }
    } else {
        thisPoint->connect(newPoint);
        allPoints->push_back(newPoint);
    }
    return collision;
}



void propagate(Point* thisPoint, float prevAngle, float bendEnergy,
              vector<Point*>* allPoints, vector<Point*>* deadEnds) {

    // Chance of an airbubble is 10%
    int chance = rand() % 20;
    bool airBubble = false;
    if (chance == 1 && bendEnergy >= 0.1){
        airBubble = true;
    }

    int numNewCracks = 1;
    float newAngle = 0.0f;
    float newX = 0.0f;
    float newY = 0.0f;
    float perfectAngle = 0.0f;
    float maxDifference = 0.0f;

    // Temp variables used in loops below
    Point* tempPoint;
    bool collision;

    if (airBubble) {
        // Randomize how many “new” cracks that should start from the
        // airbubble point.
        if (bendEnergy < 0.2){
            // Make sure there is no airbubble at the end of a crack,
            // for esthetic reasons only.
            numNewCracks = 1;
        }
        else {
```

```
        numNewCracks = 1 + rand() % (MAX_BUBBLE_CRACKS - 1);
    }
    perfectAngle = 360 / (numNewCracks + 1);
    maxDifference = (360 / (numNewCracks + 1)) -
                    (MIN_DIFFERENCE_ANGLE / 2);

}
for (int i = 1; i <= numNewCracks; i++) {


    if (airBubble) {
        //if an air bubble, the new angles are randomized fully,
        //except that they are not allowed to be closer than
        //MIN_DIFFERENCE_ANGLE to each other
        newAngle = perfectAngle * i + (fmod(rand(), maxDifference) -
                                       maxDifference / 2);

        newAngle += prevAngle + 180.0f;

        // In the current implementation there is a bug that causes
        // cracks of angles that are multiples of 90 degrees to
        // somehow bypass the collision detection algorithm. Therefore,
        // the following check is made, just to make sure no angles
        // are allowed to be an exact multiple of 90 degrees.
        if (newAngle == 0 ||
            newAngle == 90 ||
            newAngle == 180 ||
            newAngle == 270){


            newAngle += 0.1;
        }
    }
    else {

        // If not an air bubble, the angle is randomized in a narrow
        // interval
        newAngle = rand() % MIN_DIFFERENCE_ANGLE -
                            MIN_DIFFERENCE_ANGLE / 2;
        newAngle += prevAngle ;

        // Same check as above
        if (newAngle == 0 ||
            newAngle == 90 ||
            newAngle == 180 ||
            newAngle == 270){

            newAngle += 0.1;
        }
    }

    newAngle = fmod((newAngle + 360.0f), 360.0f);

    // Calculate the coordinates for the new point/points
    newX = STEPSIZE * cos(newAngle * PI / 180.0f);
    newY = STEPSIZE * sin(newAngle * PI / 180.0f);
    newX += thisPoint->getX();
    newY += thisPoint->getY();

    tempPoint = new Point(newX, newY);
    collision = safeConnect(thisPoint, tempPoint, allPoints);

    if (collision) {
        // In the current implementation, there is no phase where
        // the loose pieces actually fall off the window pane. One could
        // imagine that "spare energy" that is left when a crack
        // joins with another somehow affects how the piece that may
        // result is hurled away. In that case, the calculation of that
        // energy should probably be performed here.
    }
    else {
        if (bendEnergy / numNewCracks - ENERGY_LOSS >= 0) {
            // If the energy in the crack is not zero after
            // decreasing it, propagate is run again, with less energy.
            propagate(tempPoint, newAngle,
```

```
                            bendEnergy / numNewCracks - ENERGY_LOSS, allPoints,
                            deadEnds);
            }
            else {
                deadEnds->push_back(tempPoint);
            }
        }
    }

}

void createCrack(CrackStarter* starter,
                 vector<Point*>* allPoints,
                 vector<Point*>* deadEnds) {

    // Position the first point and add it to the main Point vector
    Point* startPoint = new Point(starter->getX(), starter->getY());
    allPoints->push_back(startPoint);

    // Temp variables used below
    float crackCosAngle;
    float crackSinAngle;
    float tempDist;
    float tempX;
    float tempY;
    Point* tempPoint;

    if (starter->getRadialDistance() != NULL) {

        // Create vector for start points of each bending-stress crack
        vector<Point*>* bendStarts = new  vector<Point*>;

        // Create vector for holding the impact-stress cross-crack points
        vector<vector<Point*>* >* impactPoints =
                                               new vector<vector<Point*>* >;

        // Position the impact-stress cross-crack points and bending-stress
        // start points
        for (int crack = 0; crack < starter->getAngles()->size(); crack++) {

            // Check if this radial crack has no cross-cracks, if so
            // continue with the next crack
            if (starter->getRadialDistance()->at(crack) == NULL) {
                // Place bend-start (extra) point but skip cross cracks
                impactPoints->push_back(NULL);

                crackCosAngle = cos(starter->getAngles()->at(crack) *
                                PI / 180.0f);
                crackSinAngle = sin(starter->getAngles()->at(crack) *
                                PI / 180.0f);

                tempDist = starter->getSafeDistance()->at(crack) + 0.01;
                tempX = tempDist * crackCosAngle;
                tempY = tempDist * crackSinAngle;
                tempX += starter->getX();
                tempY += starter->getY();

                tempPoint = new Point(tempX, tempY);
                bendStarts->push_back(tempPoint);
                startPoint->connect(tempPoint);
                allPoints->push_back(tempPoint);

                continue;
            }

            impactPoints->push_back(new vector<Point*>);

            crackCosAngle = cos(starter->getAngles()->at(crack) *
                            PI / 180.0f);
            crackSinAngle = sin(starter->getAngles()->at(crack) *
                            PI / 180.0f);

            for (int p = 0; p < starter->getRadialDistance()->at(
                                crack)->size(); p++) {
```

```
        tempDist = starter->getRadialDistance()->at(
                         crack)->at(p);

        tempX = tempDist * crackCosAngle;
        tempY = tempDist * crackSinAngle;
        tempX += starter->getX();
        tempY += starter->getY();

        tempPoint = new Point(tempX, tempY);
        impactPoints->at(crack)->push_back(tempPoint);
        allPoints->push_back(tempPoint);

        if (p == 0) {
            startPoint->connect(impactPoints->at(crack)->at(p));
        }
        else {
            impactPoints->at(crack)->at(p-1)->connect(
                          impactPoints->at(crack)->at(p));
        }
    }

    // Position bend start point
    if (starter->getRadialDistance()->at(crack)->size() > 0) {

        int pos = starter->getRadialDistance()->at(crack)->size() -
                  1;
        tempDist = starter->getRadialDistance()->at(crack)->at(pos);
        tempDist += starter->getSafeDistance()->at(crack) + 0.01;
        tempX = tempDist * crackCosAngle;
        tempY = tempDist * crackSinAngle;
        tempX += starter->getX();
        tempY += starter->getY();

        tempPoint = new Point(tempX, tempY);
        bendStarts->push_back(tempPoint);
        impactPoints->at(crack)->at(pos)->connect(tempPoint);
        allPoints->push_back(tempPoint);
    }

}

// Temp variables used below
float relAngle;
float compAngle;
float thisAngle;
float tempCos;
float tempSin;
float length;

// Position all impact-stress cross-cracks. These always start from
// the predecided points along the radial cracks and take off to the
// right, until they join with the nearest radial crack.
for (int crack = 0; crack < starter->getAngles()->size(); crack++) {

    // Check if this radial crack has no cross-cracks, if so
    // continue with the next crack
    if (starter->getRadialDistance()->at(crack) == NULL) {
        continue;
    }

    // Aquire index of the nearest radial crack, clockwise
    // orientation
    int prevCrack = crack == 0 ? starter->getAngles()->size() - 1 :
                                 crack - 1;

    // Relative angle between this radial crack and the one to the
    // right
    relAngle = starter->getAngles()->at(crack) -
                     starter->getAngles()->at(prevCrack);

    relAngle = relAngle >= 0 ? relAngle : 360.0f + relAngle;

    // Only create the cross-crack if the angle between the radial
    // cracks is less than or equal to 90 degrees
    if (relAngle <= 90.0f) {
```

```
            // Calculate cross-crack global angle
            compAngle = (180.0f - relAngle) / 2.0f;
            thisAngle = compAngle + starter->getAngles()->at(
                                  crack) + 180.0f;
            thisAngle = fmod(thisAngle + 360.0f, 360.0f);

            if(thisAngle == 0 || thisAngle == 90 ||
               thisAngle == 180 || thisAngle == 270) {
                   thisAngle -= 0.1f;
            }

            tempCos = cos(thisAngle * PI / 180.0f);
            tempSin = sin(thisAngle * PI / 180.0f);

            // The length of the cross-crack is initially set to 1.5
            // times the distance from the cross-crack start point to
            // the original impact point. This guarantees the
            // cross-crack to always join with the neighbouring radial
            // crack to the right.
            length = 1.5f * (starter->getRadialDistance()->at(
                               crack)->at(starter->getRadialDistance()->at(
                               crack)->size() - 1));

            tempX = 0.0f;
            tempY = 0.0f;

            for (int p = 0; p < starter->getRadialDistance()->at(
                crack)->size(); p++) {

                tempX = length * tempCos;
                tempY = length * tempSin;
                tempX += impactPoints->at(crack)->at(p)->getX();
                tempY += impactPoints->at(crack)->at(p)->getY();
                tempPoint = new Point(tempX, tempY);

                safeConnect(impactPoints->at(crack)->at(p), tempPoint,
                                             allPoints);

            }
        }
    }

    // Now proceed by propagating cracks caused by bending-stress
    // from all the previously defined start points.
    for (int i = 0; i < bendStarts->size(); i++) {
         if (starter->getBendingEnergyVector()->at(i) != 0){
            propagate(bendStarts->at(i),
                      starter->getAngles()->at(i),
                      starter->getBendingEnergyVector()->at(i),
                      allPoints,
                      deadEnds);
        }
    }
}
else {

    // Temp variables used below
    float startCosAngle;
    float startSinAngle;
    float tempStartX;
    float tempStartY;
    Point* tempStart;

    // Create start points for bending-stress cracks
    vector<Point*>* bendStarts = new vector<Point*>;

    for (int i = 0; i < starter->getAngles()->size(); i++) {
        if (starter->getBendingEnergyVector()->at(i) != 0){
            startCosAngle = cos(starter->getAngles()->at(i) *
                                      PI / 180.0f);
            startSinAngle = sin(starter->getAngles()->at(i) *
                                      PI / 180.0f);
            tempStartX = STEPSIZE * startCosAngle;
            tempStartY = STEPSIZE * startSinAngle;
            tempStartX += starter->getX();
```

```
                tempStartY += starter->getY();
                tempStart = new Point(tempStartX, tempStartY);
                startPoint->connect(tempStart);
                bendStarts->push_back(tempStart);
                allPoints->push_back(tempStart);
            }
        }

        // Propagate cracks caused by bending-stress, with start point in
        // the impact point (because of no impact-stress radial/cross
        // cracks).
        for (int i = 0; i < starter->getAngles()->size(); i++) {

            propagate(bendStarts->at(i),
                       starter->getAngles()->at(i),
                      starter->getBendingEnergyVector()->at(i),
                       allPoints, deadEnds);

        }
     }
  }
}
```

```
/* CrackStarter.h - This is a class that contains functions that test if
 * there will be any cracks, if so, how many and to what direction they will
 * propagate. It also have functions that give each crack circumferential cracks
 * at given distances.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */



#ifndef SPLITTER_CRACKSTARTER
#define SPLITTER_CRACKSTARTER

#include <vector>
using namespace std;

namespace splitter{
    class CrackStarter{
    public:
         CrackStarter();

         /* Initialize the CrackStarter that calls other functions that
          * calculates the crack properties. The input parameters are stored
          * globaly.
          * Input:  (hitsize) The amount of energy added to the glas.
          *         (placeX, placeY) The coordinates of the impact
          *         (CornerOneX, CornerOneY, CornertTwoX, CornerTwoY) The
          *         coordinates of two diagonal corners.
          * Output: void
          */
         void crackTester(float hitSize, float placeX, float placeY,
                          float CornerOneX, float CornerOneY, float CornerTwoX,
                          float CornerTwoY);

         /* Returns the bending energy.
          * Input:  void
          * Output: vector<float>*
          */
         vector<float>* getBendingEnergyVector();

         /* Returns a vector containing the different cracks and their
          * distances to the hitpoint.
          * Input:  void
          * Output: vector<vector<float>* >* Returns a pointer to a vector
          *         containing pointers to other vectors
          */
         vector<vector<float>* >* getRadialDistance();

         /* Returns the vector that contains the crack angle for the different
```

```
     * cracks.
     * Input:  void
     * Output: vector<float>* Returns a pointer to a vector
     */
    vector<float>* getAngles();

    /* Returns the vector with the difference between the last radial
     * cracks.
     * Input:  void
     * Output: float
     */
    vector<float>* getSafeDistance();

    /* Returns the coordinates for the impact, X and Y respectively.
     * Input:  void
     * Output: float
     */
    float getX();
    float getY();

private:

    /* Using a Weibull function to calculate the crack limit for the glas.
     * Input:  void
     * Output: float
     */
    float configureBreakLimit();

    /* Calculates and returns the real breaklimit for the glas,
     * considering the position of the impact.
     * Input:  (hitsize) The amount of energy added to the glas.
     *         (placeX, placeY) The coordinates of the impact.
     *         (CornerOneX, CornerOneY, CornertTwoX, CornerTwoY) The
     *         coordinates of two diagonal corners.
     * Output: float
     */
    float realBreakLimit(float breakLimit, float placeX, float placeY,
                         float cornerOneX, float cornerOneY,
                         float cornerTwoX, float cornerTwo);

    /* Calculates the number of cracks, considering the hitsize and the
     * real breaklimit. The number of cracks is stored in the global
     * variable numberOfCracks. Zero cracks means that the glas didn't
     * break.
     * Input:  (hitsize) The amount of energy added to the glas.
     *         (actualBreakLimit) The breaklimit for the glas.
     * Output: void
     */
    void crackGenerator(float hitsize, float actualBreakLimit);

    /* Calculates the angles for the different cracks. The angle between
     * two cracks is at least 12 degrees. When one angle is calculated,
     * another cracks gets this angle + 180 degrees to create an oposing
     * crack.
     * Input:  (angleVector) The vector that will contain the angles.
     * Output: void
     */
    void angleGenerator(vector<float>* angleVector);

    /* Creates an energy for each crack. Uses some randomisation. The
     * energy is divided into two different energies, bending energy and
     * impact energy.
     * Input:  (bendingEnergyVector) The vector that the bending energy is
     *         stored in.
     *         (impactEnergyVector) The vector that the impact energy is
     *         stored in.
     * Output: void
     */
    void energyGenerator(vector<float>* bendingEnergyVector,
                         vector<float>* impactEnergyVector);

    /* Creates a vector with distances to each radialcrack for each crack
     * and a vector with a safe distance to next crack.
     * Input:  (radialCrackVector) A pointer to a vector that for each
     *         crack containing points to other vectors in which the radial
```

```
         *         distances is stores.
         *         (safeDistance) A pointer to a vector containing safe
         *         distances to next crack.
         * Output: void
         */
        void radialCrackGenerator(vector<vector<float>* >*& radialCrackVector,
                                  vector<float>*& safeDistance);


        int weibull;                  //A number between 1-20, every number
                                      //symbols five percent.
        float breakLimit;            //Our windows break limit in the centre
                                      //of the window.
        float actualBreakLimit;      //Break limit where the hit stroke.
        float centerX;               //Our windows center point.
        float centerY;
        float distanceToCenter;      //The distance from the hit to the center
        float maxDifference;         //The maximum difference between angles
        float placeX;                //Hit position.
        float placeY;
        float BREAK_LIMIT;           //how much the position change the break
                                      //limit.
        float CRACK_NUMBER;          //How many cracks will errupt due to the
                                      //energy.
        float ENERGY_DIFFER;         //How much will the energy differ between
                                      //the cracks.
        int NORMAL_BREAK;             //A window normal break limit.
        int BENDING_INTERVAL;         //The energy interval where only bending
                                      //accurs.
        int CROSS_INTERVAL;           //The energy interval where both bending
                                      //and impact accurs.
        float RADIAL_FACTOR;         //Decides the number of cracks.
        int MAX_RADIALS;              //Maximum number of radials.
        float DISTANCE_FACTOR;       //A distance factor to determine the
                                      //radial cracks.
        int MAX_CRACKS;               //The maximun number of cracks.
        int MIN_CRACK_DIFFERENCE;     //minimum difference between the cracks.
        vector<float>* angleVector;
        vector<float>* safeDistance;
        vector<float>* bendingEnergyVector;
        vector<float>* impactEnergyVector;
        vector<vector<float>* >* radialCrackVector;
        int numberOfCracks;
        float crackEnergy;

    };
}
#endif
```

```
/* CrackStarter.cpp - This is a class that contains functions that test if
 * there will be any cracks, if so, how many and to what direction they will
 * propagate. It also have functions that give each crack circumferential cracks
 * at given distances.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */



#include "CrackStarter.h"
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <vector>
using namespace std;

namespace splitter{

    CrackStarter::CrackStarter() {         //initials the constants
        BREAK_LIMIT = 1;
        CRACK_NUMBER = 1;
        ENERGY_DIFFER = 0.099;
        NORMAL_BREAK = 10;
        BENDING_INTERVAL = 5;
        CROSS_INTERVAL = 20;
        RADIAL_FACTOR = 20.0f;
        MAX_RADIALS = 10;
        DISTANCE_FACTOR = 3;
        MAX_CRACKS = 30;
        MIN_CRACK_DIFFERENCE = 12;

        angleVector = new vector<float>;
        bendingEnergyVector = new vector<float>;
        impactEnergyVector = new vector<float>;
        radialCrackVector = new vector<vector<float>* >;
        safeDistance = new vector<float>;
    }

    void CrackStarter::crackTester(float hitSize,
                                   float placeX, float placeY,
                                   float cornerOneX, float cornerOneY,
                                   float cornerTwoX, float cornerTwoY) {

        this->placeX = placeX;
        this->placeY = placeY;

        actualBreakLimit = realBreakLimit(configureBreakLimit(),
                                          placeX, placeY,
```

```
                                    cornerOneX, cornerOneY,
                                    cornerTwoX, cornerTwoY);

    crackGenerator(hitSize, actualBreakLimit);

    if (numberOfCracks != 0) {
        angleGenerator(angleVector);
        energyGenerator(bendingEnergyVector, impactEnergyVector);
        radialCrackGenerator(radialCrackVector, safeDistance);
    }
}

float CrackStarter::configureBreakLimit() {

    weibull = 1 + rand() % 20;

//Decides what breakLimit will be, this is
//different from time to time
    switch (weibull) {
      case 1: {
          breakLimit = NORMAL_BREAK*0.55;
          break;
      }
      case 2: {
          breakLimit = NORMAL_BREAK*0.65;
          break;
      }
      case 3: {
          breakLimit = NORMAL_BREAK*0.70;
          break;
      }
      case 4: {
          breakLimit = NORMAL_BREAK*0.75;
          break;
      }
      case 5: {
          breakLimit = NORMAL_BREAK*0.80;
          break;
      }
      case 6: {
          breakLimit = NORMAL_BREAK*0.83;
          break;
      }
      case 7: {
          breakLimit = NORMAL_BREAK*0.85;
          break;
      }
      case 8: {
          breakLimit = NORMAL_BREAK*0.87;
          break;
      }
      case 9: {
          breakLimit = NORMAL_BREAK*0.89;
          break;
      }
      case 10: {
          breakLimit = NORMAL_BREAK*0.92;
          break;
      }
      case 11: {
          breakLimit = NORMAL_BREAK*0.95;
          break;
      }
      case 12: {
          breakLimit = NORMAL_BREAK*0.97;
          break;
      }
      case 13: {
          breakLimit = NORMAL_BREAK*1.01;
          break;
      }
      case 14: {
          breakLimit = NORMAL_BREAK*1.03;
          break;
      }
```

```
        case 15: {
            breakLimit = NORMAL_BREAK*1,05;
            break;
        }
        case 16: {
            breakLimit = NORMAL_BREAK*1.07;
            break;
        }
        case 17: {
            breakLimit = NORMAL_BREAK*1.10;
            break;
        }
        case 18: {
            breakLimit = NORMAL_BREAK*1.14;
            break;
        }
        case 19: {
            breakLimit = NORMAL_BREAK*1.20;
            break;
        }
        case 20: {
            breakLimit = NORMAL_BREAK*1.30;
            break;
        }
    }

return breakLimit;
}

//This changes the break limit if you don't hit in the center of the window.
float CrackStarter::realBreakLimit(float breakLimit,
                                   float placeX, float placeY,
                                   float cornerOneX, float cornerOneY,
                                   float cornerTwoX, float cornerTwoY) {

    centerX = (cornerTwoX - cornerOneX) / 2 + cornerOneX;
    centerY = (cornerOneY - cornerTwoY) / 2 + cornerOneY;
    distanceToCenter = pow((centerX - placeX), 2) + pow((centerX - placeX),
                       2);

    actualBreakLimit = breakLimit - (BREAK_LIMIT * sqrt(distanceToCenter));

    return actualBreakLimit;
}

void CrackStarter::crackGenerator(float hitsize, float actualBreakLimit) {

    crackEnergy = hitsize - actualBreakLimit;

    if (crackEnergy <= 0){
        numberOfCracks = 0;
    }
    else{
        //There should always be an even number of cracks.
        numberOfCracks = 2 * int(crackEnergy*CRACK_NUMBER);
        if (numberOfCracks > MAX_CRACKS) {
            numberOfCracks = MAX_CRACKS;
        }
    }
}

void CrackStarter::angleGenerator(vector<float>* angleVector) {

    float perfectAngle = 360 / numberOfCracks;
    int maxDifference = (360 / numberOfCracks) - MIN_CRACK_DIFFERENCE / 2 ;
    float angle;

    for (int i = 0; i < (numberOfCracks/2); i++) {

        // Give each crack an angle.
        angle = maxDifference/2 + perfectAngle * i +
                (rand() % maxDifference - maxDifference / 2);

        if (angle == 0||angle == 90||angle == 180||angle == 270){
            angle += 0.1;    // Due to the multiple-of-90-degrees-bug.
```

```
        }

        angleVector->push_back(angle);
    }

    for (int i = 0; i < (numberOfCracks / 2); i++) {
        // Cracks continue in both directions.
        angleVector->push_back(angleVector->at(i) + 180);
    }
}

//Divides the energy to two differnt types of energy. One that decides how
//mant circumferential crack there will be and one that decides for how long
//the crack will go on after them.
void CrackStarter::energyGenerator(vector<float>* bendingEnergyVector,
                                   vector<float>* impactEnergyVector) {

    float totalBendingEnergy;
    float totalImpactEnergy;
    float relationEnergy = (crackEnergy - BENDING_INTERVAL) /
                           CROSS_INTERVAL;

    if (relationEnergy <= 0) {
        totalBendingEnergy = crackEnergy;
        totalImpactEnergy = 0;
    }
    else if (relationEnergy > 0 && relationEnergy < 1) {
        totalImpactEnergy = crackEnergy * relationEnergy;
        totalBendingEnergy = crackEnergy - totalImpactEnergy;
    }
    else {
        totalImpactEnergy = crackEnergy;
        totalBendingEnergy = 0;
    }

    float impactEnergyPerCrack = totalImpactEnergy / numberOfCracks;
    float bendingEnergyPerCrack = totalBendingEnergy / numberOfCracks;

    for (int i = 0; i < numberOfCracks; i++) {

        if (impactEnergyPerCrack > 0) {
            // Randomize the impact energy a bit
            impactEnergyVector->push_back(impactEnergyPerCrack + (fmod(
                                    rand(),
                                    ENERGY_DIFFER) - ENERGY_DIFFER / 2));
        }
        else{
            impactEnergyVector->push_back(0);
        }

        if (bendingEnergyPerCrack > 0) {
           // Randomize the bending energy a bit
           float tempEnergy= bendingEnergyPerCrack + (fmod(
                             rand(), ENERGY_DIFFER) - ENERGY_DIFFER / 2);

           if (tempEnergy < 0){
               bendingEnergyVector->push_back(0);
           }
           else{
               bendingEnergyVector->push_back(tempEnergy);
           }

        }
        else {
            bendingEnergyVector->push_back(0);
        }
    }
}

//Creates the circumferential cracks distance to the impact point.
void CrackStarter::radialCrackGenerator(
                               vector<vector<float>* >*& radialCrackVector,
                               vector<float>*& safeDistance) {

    float minLast = 0;
```

```
float maxLast = 0;
int zeroRadials = 0;
vector<float>* radialVector;

for (int i = 0; i < numberOfCracks; i++) {
    if (impactEnergyVector->at(i) <= 0.0f) {  //if there is no energy
        radialCrackVector->push_back(NULL);   //there will be no cracks.
        zeroRadials++;

    }
    else {

        float radialEnergy = (impactEnergyVector->at(i) *
                              RADIAL_FACTOR);

        int numberOfRadials = (int)radialEnergy;

        if (numberOfRadials > MAX_RADIALS) {
            numberOfRadials = MAX_RADIALS;
        }

        float firstDistance = DISTANCE_FACTOR / radialEnergy;

        if (firstDistance > 2.5){
            firstDistance = 2.5;
        }

        radialVector = new vector<float>;

        if (numberOfRadials > 0){

            radialVector->push_back(firstDistance);

            if (numberOfRadials > 1){
                for (int j = 1; j < numberOfRadials; j++) {
                    radialVector->push_back(radialVector->at(j-1) +
                                            firstDistance /
                                            pow(2.0, (double)j));
                }
            }
        }
        else {
            radialVector = NULL;
            zeroRadials++;
        }

        radialCrackVector->push_back(radialVector);
    }
}

if (zeroRadials == numberOfCracks){
    radialCrackVector = NULL;
}

else {
    float thisLast;
    float nextLast;

    for (int i = 0; i<numberOfCracks - 1; i++) {

        if (radialCrackVector->at(i) != NULL) {
            thisLast = radialCrackVector->at(i)->at(
                                    radialCrackVector->at(i)->size()-1);
        }
        else {
            thisLast = 0.0f;
        }
        if (radialCrackVector->at(i + 1) != NULL) {
            nextLast = radialCrackVector->at(i + 1)->at(
                              radialCrackVector->at(i + 1)->size() - 1);

        }
        else {
            nextLast = 0.0f;
        }
```

```
                if ((nextLast - thisLast) >= 0){
                    safeDistance->push_back(nextLast - thisLast);
                }
                else {
                    safeDistance->push_back(0);
                }
            }

            if (radialCrackVector->at(numberOfCracks - 1) != NULL) {
                thisLast = radialCrackVector->at(numberOfCracks - 1)->at(
                         radialCrackVector->at(numberOfCracks - 1)->size() - 1);
            }
            else {
                thisLast = 0.0f;
            }
            if (radialCrackVector->at(0) != NULL) {
                nextLast = radialCrackVector->at(0)->at(
                                          radialCrackVector->at(0)->size() - 1);
            }
            else {
                nextLast = 0.0f;
            }
            if ((nextLast - thisLast) >= 0){
                safeDistance->push_back(nextLast - thisLast);
            }
            else {
                safeDistance->push_back(0);

            }
        }

    }

    vector<float>* CrackStarter::getAngles() {
        if (numberOfCracks == 0) {
            return NULL;
        }
        return angleVector;
    }

    vector<float>* CrackStarter::getBendingEnergyVector() {
        return bendingEnergyVector;
    }

    vector<float>* CrackStarter::getSafeDistance() {
        return safeDistance;
    }

    float CrackStarter::getX() {
        return placeX;
    }
    float CrackStarter::getY() {
        return placeY;
    }

    vector<vector<float>* >* CrackStarter::getRadialDistance() {
        return radialCrackVector;
    }
}
```

```
/* graphics.h - File to handle all graphical parts except for the tesselation
 * of the glass polygon, done in tesselate.cpp. graphics.h is included in
 * splitter.cpp and needs point.h to run.
 *
 * Authors:
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <windows.h>
#include <GL/glfw.h>
#include “Point.h”

using namespace std;

namespace splitter {

    //Data types for the bitmap fonts
    typedef struct {
        GLuint base;        //Display list number of first character
        int    widths[256]; //Width of each character in pixels
        int    height;      //Height of characters
    } GLFONT;

    /* Initialize and set up the graphical environment.
     *
     * Input:  void
     * Output: void
     */
    void initGraphics(int resWidth, int resHeight);

    /* Initialize the allPoints and cs objects, and set the clickedAlready
     * flag to false (so that it is possible to create new cracks). This
     * removes all previous cracks.
     *
     * Input:  void.
     * Output: void.
     */
    void resetCracks();

    /* Initialize and set up the input dependent user interface.
     *
     * Input:  void
     * Output: void
     */
    void initUIInput();

    /* Calculate and report frames per second
     * (updated once per second) in the window title bar.
     *
     * Input:  void
```

```
 * Output: void
 */
void showFPS();

/* Calculates the time since the last frame.
 * This is used to update the inputs from the keyboards.
 * Changes the variable frameTime.
 *
 * Input:  void
 * Output: void
 */
void updateFrameTime();

/* Get the mouse position from the screen.
 *
 * Input:  void
 * Output: void
 */
void updateMousePos();

/* Uppdate horizontal movment (xz-plane) and view direction.
 *
 * Input:  void
 * Output: void
 */
void updateDirPos();

/* Gets the cooridinate of the point that is projected on the screen
 * coordinates, (x,y).
 *
 * Input:  (x) screen x - coordinate
 *         (y) screen y - coordinate
 * Output: void
 */
void getOGLPos(int x, int y, GLfloat& posX, GLfloat& posY, GLfloat& posZ);

/* Uppdate the impact velocity applied from the user with the + and -
 * numpad keys.
 *
 * Input:  void
 * Output: void
 */
void updateImpactVelocity();

/* Calls function getOGLPos to check if pointing on the glass window.
 * Runs the function crackTester to se if the glass breaks. Also runs
 * createCrack to calculate the crackPattern.
 * Only possible to press the button on one side off the wall.
 *
 * Input:  (button) the specified button asigned for the task
 *         (action) is the button pressed or released
 * Output: void
 */
void GLFWCALL updateCrackPattern(int button, int action);

/* Create a bitmap font.
 *
 * Input:  (hdc) Device Context
 *         (*typeface) Font specification
 *         (height) Font height/size in pixels
 *         (weight) Weight of font (bold, etc)
 *         (italic) Text is italic
 * Output: (GLFONT) The font
 */
GLFONT * fontCreate(HDC hdc, const char *typeface,
                    int height, int weight, DWORD italic);

/* Delete the specified font.
 *
 * Input:  (*font) Font to delete
 * Output: void
 */
void fontDelete(GLFONT *font);

/* Display a string using the specified font.
```

```
     *
     * Input:  (*font) Font to use
     *         (*s) String to display
     * Output: void
     */
    void fontPuts(GLFONT *font,const char *s);

    /* Display a formatted string using the specified font.
     *
     * Input:  (*font) Font to use
     *         (align) Alignment to use
     *         (*format) printf() style format string, i.e. text to print
     *         (...) Other arguments as necessary
     * Output: void
     */
    void fontPrintf(GLFONT *font, int align, char *format, ...);

    /* Draw all the graphics to the buffer.
     *
     * Input:  void
     * Output: void
     */
    void draw();

    /* Draw the crack pattern.
     *
     * Input:  (allPoints) the vector containing all points and their
     *         connections
     * Output: void
     */
    void drawCracks(vector<Point*>* allPoints);

    /* Configure camera options.
     *
     * Input:  (w) widht of window
     *         (h) height of window
     * Output: void
     */
    void GLFWCALL reshape(int w, int h);

    /* Check keyboard inputs, arrow keys and ‘w’ ‘a’ ‘s’ ‘d’.
     *
     * Input:  (key) specifies a key
     *         (action) checks if the specified key is pressed or relesed
     * Output: void
     */
    void GLFWCALL keyPressed(int key, int action);

    /* Deinitialize all graphical elements.
     *
     * Input:  void
     * Output: void
     */
    void deinitGraphics();
}
```

```
/* graphics.cpp - File to handle all graphical parts except for the tesselation
 * of the glass polygon, done in tesselate.cpp. graphics.cpp is executed from
 * splitter.cpp and needs to include graphics.h, room.h, carackCreation.h,
 * crackStarter.h, Point.h, holes.h and tesselate.h to run.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <windows.h>
#include <GL/glfw.h>
#include <math.h>
#include <iostream>
#include <vector>
#include “room.h”
#include “graphics.h”
#include “crackCreation.h”
#include “CrackStarter.h”
#include “Point.h”
#include “holes.h”
#include “tesselate.h”

using namespace std;

namespace splitter {

    // Point related
    CrackStarter* cs;
    vector<Point*>* allPoints;
    vector<Point*>* deadEnds;
    vector<vector<Point*>* >* holes;


    // Checks if the left resp. the right mouse button has been pressed
    bool mlClickedAlready = false;
    bool mrClickedAlready = false;

    // Define camera position
    float pos[] = {0.0, 6.0, 6.0};

    // Constants
    // Moving parameters

    const float MOUSE_SENSITIVITY = 0.1f;
    const float WALKING_SPEED = 10.0f;

    // Convertion degrees to radians
    const float DEG2RAD = 3.14159265 / 180.0;

    // Window size
```

```
int WIDTH = 1024;
int HEIGHT = 768;

// Impact velocity
float impactVelocity = 0.0f;

// For the fps
double t0 = 0.0;
int frames = 0;
char titlestring[200];

// To get the frametime
float frameTime = 0;
double tOld = 0;

// Angles for the view-vector
float hAngle = 0.0;
float vAngle = 0.0;

// Keyboard inputs
bool upPressed = false;
bool downPressed = false;
bool leftPressed = false;
bool rightPressed = false;
bool plusPressed = false;
bool minusPressed = false;

// Mouse coordinates
int mouseX = 0;
int mouseY = 0;
int mouseOldX = 0;
int mouseOldY = 0;

// Projection
GLfloat posX, posY, posZ;

// Font data
GLFONT *font;

// Ambient light
GLfloat globalLight[] = {0.5, 0.5, 0.5, 1.0};

// Room display list index
GLuint roomList;

void initGraphics(int resW, int resH) {

    WIDTH = resW;
    HEIGHT = resH;

    glfwInit();
    glfwOpenWindow(WIDTH, HEIGHT, 8, 8, 8, 8, 24, 0, GLFW_FULLSCREEN);
    glfwSetWindowPos(200, 200);
    glfwDisable(GLFW_MOUSE_CURSOR);
    glfwEnable(GLFW_STICKY_KEYS);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_SMOOTH);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, globalLight);
    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_NORMALIZE);

    // Initialize textures
    initTextures();

    // Init room display list
   roomList = initRoomDisplayList();

    // Used to enable blending.
    glEnable(GL_COLOR_MATERIAL);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    // Used to prevent aliasing (in blending)
```

```
    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_POINT_SMOOTH);

    // Create a new font
    font = fontCreate(wglGetCurrentDC(), "Verdana", 15, 0, 0);

    // Init point vector and CrackStarter object
    resetCracks();
}

void resetCracks() {

    if (allPoints != NULL) {
        // Delete all points in the vector, and the vector.
        for (int i = 0; i < allPoints->size(); i++) {
            delete allPoints->at(i);
        }
        delete allPoints;
    }

    if (deadEnds != NULL) {
        // Delete the vector
        delete deadEnds;
    }

    if (holes != NULL) {
        // Delete the different vectors including the hole points.
        for (int i = 0; i < holes->size(); i++) {
            delete holes->at(i);
        }
        delete holes;
    }

    holes = new vector<vector<Point*>* >;

    // Create a new vector including all the points and the points marked as
    // dead ends.
    allPoints = new vector<Point*>;
    deadEnds = new vector<Point*>;

    if (cs != NULL) {
        delete cs;
    }

    // Create a new object that determents the crack pattern.
    cs = new CrackStarter();

    resetTessList();

    mlClickedAlready = false;
    mrClickedAlready = false;
}

void initUIInput() {
    glfwSetWindowSizeCallback(reshape);
    glfwSetKeyCallback(keyPressed);
    glfwSetMouseButtonCallback(updateCrackPattern);
    glfwEnable(GLFW_STICKY_KEYS);
}

void showFPS() {

    double fps;

    // Get current time
    double t = glfwGetTime();  // Gets number of seconds since glfwInit()
    // If one second has passed, or if this is the very first frame
    if ((t-t0) > 1.0 || frames == 0) {
        fps = (double)frames / (t-t0);
        sprintf(titlestring, "World (%.1f FPS)", fps);
        glfwSetWindowTitle(titlestring);
        t0 = t;
        frames = 0;
    }
    frames ++;
```

```
}


void updateFrameTime() {
    double t = glfwGetTime();
    frameTime = (float)(t - tOld);
    tOld = t;
}


void updateMousePos() {
    int x, y;
    glfwGetMousePos(&x, &y);
    mouseOldX = mouseX;
    mouseX = x;
    mouseOldY = mouseY;
    mouseY = y;
}

void updateDirPos() {
    float xFactor = (mouseX-mouseOldX) * MOUSE_SENSITIVITY;
    float yFactor = (mouseY-mouseOldY) * MOUSE_SENSITIVITY;

    if (yFactor + vAngle <= 80.0 && yFactor + vAngle >= -80.0) {
        vAngle = yFactor + vAngle;
    }

    hAngle += xFactor;

    if (hAngle >= 360.0) {
        hAngle -= 360.0;
    }
    if (hAngle < 0.0) {
        hAngle = 360.0 + hAngle;
    }
    if (pos[0] <= 13 && pos[0] >= -13 && pos[2] >= -13 && pos[2] <= 13){
        if (pos[0] < 11 && pos[0] > -11 && pos[2] > -2 && pos[2] < 2){
           // Inside the brick wall, not allowed to walk.
           if (pos[2] >= 0) {
                pos[2] = 2;
            }
           else if (pos[2] < 0) {
                pos[2] = -2;
            }

     }
        else {
            if (upPressed) {
                pos[0] += frameTime * WALKING_SPEED * sin(DEG2RAD*hAngle);
                pos[2] -= frameTime * WALKING_SPEED * cos(DEG2RAD*hAngle);
            }
            if (downPressed) {
                pos[0] -= frameTime * WALKING_SPEED * sin(DEG2RAD*hAngle);
              pos[2] += frameTime * WALKING_SPEED * cos(DEG2RAD*hAngle);
            }
            if (leftPressed) {
                pos[0] -= frameTime * WALKING_SPEED * cos(DEG2RAD*hAngle);
               pos[2] -= frameTime * WALKING_SPEED * sin(DEG2RAD*hAngle);
            }
            if (rightPressed) {
               pos[0] += frameTime * WALKING_SPEED * cos(DEG2RAD*hAngle);
               pos[2] += frameTime * WALKING_SPEED * sin(DEG2RAD*hAngle);
            }

            if (pos[0] > 13) {
               pos[0] = 13;
            }
            else if (pos[0] < -13) {
                pos[0] = -13;
            }
            if (pos[2] > 13) {
               pos[2] = 13;
        }
            else if (pos[2] < -13) {
                pos[2] = -13;
```

```
            }
        }
    }
}

void getOGLPos(int x, int y, GLfloat& posX, GLfloat& posY, GLfloat& posZ) {
    GLint viewport[4];
    GLdouble modelview[16];
    GLdouble projection[16];
    GLfloat winX, winY, winZ;
    GLdouble tempX, tempY, tempZ;

    glGetDoublev( GL_MODELVIEW_MATRIX, modelview );
    glGetDoublev( GL_PROJECTION_MATRIX, projection );
    glGetIntegerv( GL_VIEWPORT, viewport );

    winX = (float)x;
    winY = (float)viewport[3] - (float)y;
    glReadPixels( x, int(winY), 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &winZ );

    gluUnProject(winX, winY, winZ, modelview, projection, viewport,
                 &tempX, &tempY, &tempZ);
    posX = tempX;
    posY = tempY;
    posZ = tempZ;
}

void updateImpactVelocity() {
    if (plusPressed) {
        if (impactVelocity < 20.0f) {
            impactVelocity += 10.0f * frameTime;
        } else if (impactVelocity >= 20.0f && impactVelocity < 40.0f) {
            impactVelocity += 50.0f * frameTime;
        }
    }
    if (minusPressed) {
        if (impactVelocity > 0.1f && impactVelocity <= 20.0f) {
            impactVelocity -= 10.0f * frameTime;
        } else if (impactVelocity > 20.0f) {
            impactVelocity -= 50.0f * frameTime;
        }
    }
}

void GLFWCALL updateCrackPattern(int button, int action) {
    if (button == GLFW_MOUSE_BUTTON_LEFT &&
        action == GLFW_PRESS && pos[2] > 0) {

        if (mlClickedAlready == false) {

            getOGLPos(WIDTH/2, HEIGHT/2, posX, posY, posZ);
         if (posX <= 3.5 && posX >= -3.5 &&
                posY <= 8.5 && posY >= 1.5 &&
                posZ <= 0.1 && posZ >= -0.1) {

                // Test if the glass breaks, propagate cracks etc.
                cs->crackTester(impactVelocity, posX, posY, -3.5f, 8.5f,
                                3.5f, 1.5f);

                if (cs->getAngles() != NULL) {
                    mlClickedAlready = true;
                    createCrack(cs, allPoints, deadEnds);
             }
            }
        }
    }

    if (button == GLFW_MOUSE_BUTTON_RIGHT &&
        action == GLFW_PRESS && pos[2] > 0) {

        if (mlClickedAlready == true && mrClickedAlready == false) {

            findDeadEnds(deadEnds, allPoints);
            findHoles(holes, allPoints);
            mrClickedAlready == true;
```

```
        }
    }
}

GLFONT * fontCreate(HDC hdc, const char *typeface,
                    int height, int weight, DWORD italic) {

    GLFONT *font;       // Font data pointer
    font = new GLFONT;  // Memory allocation
    HFONT fontid;       // Windows font ID
    int charset;        // Character set code

    // Allocate display lists
    if ((font->base = glGenLists(256)) == 0) {
        free(font);
        return (0);
    }

    charset = ANSI_CHARSET;

    // Load the font
    fontid = CreateFont(height, 0, 0, 0, weight, italic, FALSE, FALSE,
                        charset, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                        DRAFT_QUALITY, DEFAULT_PITCH, typeface);

    SelectObject(hdc, fontid);

    // Create bitmaps for each character
    wglUseFontBitmaps(hdc, 0, 256, font->base);

    // Get the width and height information for each character
    GetCharWidth(hdc, 0, 255, font->widths);
    font->height = height;

    return (font);
}

void fontDelete(GLFONT *font) {
    if (font == (GLFONT *)0) {
        return;
    }
    glDeleteLists(font->base, 256);
    free(font);
}

void fontPuts(GLFONT *font,const char *s) {
    if (font == (GLFONT *)0 || s == NULL) {
        return;
    }
    glPushAttrib(GL_LIST_BIT);
        glListBase(font->base);
        glCallLists(strlen(s), GL_UNSIGNED_BYTE, s);
    glPopAttrib();
}

void fontPrintf(GLFONT *font, int align, char *format, ...) {
    va_list ap;         // Argument pointer
    char s[1024], *ptr; // Output string, Pointer into string
    int width;          // Width of string in pixels

    if (font == (GLFONT *)0 || format == (char *)0) {
        return;
    }

    // Format the string
    va_start(ap, format);
    vsprintf((char *)s, format, ap);
    va_end(ap);

    // Figure out the width of the string in pixels...
    for (ptr = s, width = 0; *ptr; ptr ++) {
        width += font->widths[*ptr];
    }
    // Adjust the bitmap position as needed to justify the text
    if (align < 0) {
```

```
        glBitmap(0, 0, 0, 0, -width, 0, NULL);
    }
    else if (align == 0) {
        glBitmap(0, 0, 0, 0, -width / 2, 0, NULL);
    }

    // Draw the string
    fontPuts(font, s);
}

void draw() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    updateFrameTime();
    updateMousePos();
    updateDirPos();
    updateImpactVelocity();

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Draw the crosshair centered on the screen, center alignment
    glPushMatrix();
        glColor3f(1.0f, 0.0f, 0.0f);
        glTranslatef(0.0f, 0.0f, -1.5f);
        glRasterPos2i(0,0);
        fontPrintf(font, 0, "X");
    glPopMatrix();

    // Draw text in lower right corner, left alignment
    glPushMatrix();
        glColor3f(1.0f, 0.0f, 0.0f);
        glTranslatef(1.0f, -1.0f, -1.5f);
        glRasterPos2i(0,0);
        fontPrintf(font, 1, "Impact velocity: %4.0f", impactVelocity);
    glPopMatrix();

    // Rotate and translate the world
    glRotatef(vAngle, 1.0, 0.0, 0.0);
    glRotatef(hAngle, 0.0, 1.0, 0.0);
    glTranslatef(-pos[0], -pos[1], -pos[2]);

    // Execute the room display list
    glCallList(roomList);

    if (allPoints->size() > 1) {
       drawCracks(allPoints);
    }

    drawWindow(holes);

    glFlush();
}

void drawCracks(vector<Point*>* allPoints) {

    Point* p1;
    Point* p2;
    glEnable(GL_BLEND);
    glColor4f(1.0f, 1.0f, 1.0f, 0.5f);
    glLineWidth(0.5f);
    for (int i = 0; i < allPoints->size(); i++) {

        p1 = allPoints->at(i);
        for (int j = 0; j < p1->getNumConnections(); j++) {

            p2 = p1->getPoint(j);
            if (holes->size() == 0) {
                // Draw all the cracks belonging to holes
                if (p1->getTestFlag(j) == false &&
                       p1->getEndFlag(j) == false) {

                    glPushMatrix();
                        glBegin(GL_LINES);
                            glVertex3f(p1->getX(), p1->getY(), 0.0f);
```

```
                            glVertex3f(p2->getX(), p2->getY(), 0.0f);
                        glEnd();
                    glPopMatrix();
                }
            }
        // Draw the rest of the cracks, i.e. the dead ends
            if (p1->getTestFlag(j) == false && p1->getEndFlag(j)== true) {
                glPushMatrix();
                    glBegin(GL_LINES);
                        glVertex3f(p1->getX(), p1->getY(), 0.0f);
                        glVertex3f(p2->getX(), p2->getY(), 0.0f);
                    glEnd();
                glPopMatrix();
            }

            // Switch the flag, indicating that the connection has been
            // drawn, or just resetting it.
            p1->setTestFlag(j, !(p1->getTestFlag(j)));
        }
    }
    glDisable(GL_BLEND);
}


void GLFWCALL reshape(int w, int h) {
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glfwSetMousePos(0, 0);
    mouseX = 0;
    mouseOldX = 0;
    mouseY = 0;
    mouseOldY = 0;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(
        80.0,
        (double) w/ (double) h,
        1.0,
        300
    );
}

void GLFWCALL keyPressed(int key, int action) {
    if (action == GLFW_PRESS) {
        switch (key) {
            case GLFW_KEY_UP: {
                upPressed = true;
                break;
          }
          case GLFW_KEY_DOWN: {
             downPressed = true;
                break;
          }
           case GLFW_KEY_LEFT: {
            leftPressed = true;
                break;
           }
          case GLFW_KEY_RIGHT: {
            rightPressed = true;
                break;
          }
          case 'W': {
                 upPressed = true;
                break;
           }
          case 'S': {
              downPressed = true;
                break;
          }
          case 'A': {
            leftPressed = true;
                break;
           }
           case 'D': {
            rightPressed = true;
                break;
```

```
            }
             case GLFW_KEY_KP_ADD: {
             plusPressed = true;
                 break;
         }
         case GLFW_KEY_KP_SUBTRACT: {
             minusPressed = true;
                 break;
         }
           default:
             break;
          }
     }
     if (action == GLFW_RELEASE) {
          switch (key) {
              case GLFW_KEY_UP: {
                  upPressed = false;
                 break;
            }
           case GLFW_KEY_DOWN: {
               downPressed = false;
                 break;
           }
           case GLFW_KEY_LEFT: {
                 leftPressed = false;
                 break;
           }
           case GLFW_KEY_RIGHT: {
                 rightPressed = false;
                 break;
         }
         case 'W': {
                   upPressed = false;
                 break;
           }
           case 'S': {
               downPressed = false;
                 break;
           }
           case 'A': {
                 leftPressed = false;
                 break;
           }
           case 'D': {
                 rightPressed = false;
                 break;
          }
              case GLFW_KEY_KP_ADD: {
             plusPressed = false;
                 break;
         }
         case GLFW_KEY_KP_SUBTRACT: {
                 minusPressed = false;
                 break;
         }
         case GLFW_KEY_SPACE: {
             resetCracks();
             break;
         }
         default:
                 break;
           }
     }
 }

 void deinitGraphics() {
     fontDelete(font);
     glfwTerminate();
 }

}
```

```
/* holes.h - File to find all the parts of the crack pattern belonging to a
 * dead end and function for finding all the “smallest” holes (i.e. the smallest
 * glass pieces made by the crack pattern). Included in graphics.cpp.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

using namespace std;

namespace splitter {

    /* Finds the part of the crack pattern that has propagated further after a
     * hole, or the hole crack pattern if there are no found holes. Saves the
     * found dead ends in the deadEnds vector.
     *
     * Input:  (deadEnds) a vector containing the end points of all created
     *         cracks. An end point is defined as: A point where the bending
     *         energy is zero.
     * Output: void
     */
    void findDeadEnds(vector<Point*>* deadEnds, vector<Point*>* allPoints);

    /* Finds the smallest pieces in a crack pattern, wich is all the holes (not
     * containing any other holes). Saves the points defining the holes in the
     * holes vector.
     *
     * Input:  (holes) an empty vector to store the vectors fore the holes in.
     *         (allPoints) vector containing all the points in the crack
     *         pattern.
     * Output: void
     */
    void findHoles(vector<vector<Point*>* >* holes, vector<Point*>* allPoints);

}
```

```
/* holes.cpp - File to find all the parts of the crack pattern belonging to a
 * dead end and function for finding all the "smallest" holes (i.e. the smallest
 * glass pieces made by the crack pattern). Needs to include holes.h and
 * Point.h. Executed from graphics.cpp.
 *
 * Authors:
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <vector>
#include "Point.h"
#include "holes.h"
#include <cmath>

using namespace std;

namespace splitter {

    void findDeadEnds(vector<Point*>* deadEnds, vector<Point*>* allPoints) {
         int id = -1;
         Point* p;

         int counter = 0;

         int notDeadEnd = 0;


         // Check if there are any dead ends
         if (deadEnds->size() != 0) {

            // Loop trough all dead ends
             for (int i = 0; i < deadEnds->size(); i++) {

                 id = deadEnds->at(i)->getID();
                 p = allPoints->at(id);

                 // If number of connections in one point is larger than four,
                 // it must be a hole, due to the number of possible cracks from
                 // an airbubble.
                 if (p->getNumConnections() < 5){
                     int j = p->getNumConnections();
                     while(j > 0 && j < 5) {
                         counter = 0;
                         // Checks how many connections already set to dead
                         // end. If the number of dead ends are one less than
                         // the number of connections then the connection is
                         // also a dead end.
                         for (int connection = 0; connection < p->
                              getNumConnections(); connection++) {
```

```
                        if (p->getEndFlag(connection)) {
                            counter++;
                        }
                        else {
                            notDeadEnd = connection;
                        }
                    }

                    if (counter == p->getNumConnections() - 1) {
                        p->setEndFlag(notDeadEnd, true);
                        p = p->getPoint(notDeadEnd);
                    }
                    else {
                        break;
                    }

                    j = p->getNumConnections();
                }
            }

        }

    }
}

void findHoles(vector<vector<Point*>* >* holes, vector<Point*>* allPoints) {
    bool checkedAll = false;
    float polygonAngle = 0.0f;
    float thisAngle = 0.0f;
    float nextAngle = 0.0f;
    float firstAngle = 0.0f;
    int pos = 0;
    Point* firstPoint;
    Point* thisPoint;
    Point* prevPoint;
    Point* nextPoint;
    Point* p;
    vector<Point*>* holePoints = new vector<Point*>;;

    // Loop through all points
    for (int i = 0; i < allPoints->size(); i++) {
        p = allPoints->at(i);

        // For every point check all the connections
        for (int connection = 0; connection < p->getNumConnections();
             connection++) {

            // If the connection is already visited the path is not
            // availible, go to next connection. Also checks if it's a
            // dead end.
            if (p->getPath(connection) != false &&
                p->getEndFlag(connection) == false) {

                polygonAngle = 0.0f;
                firstPoint = p;
                // Indicate that the connection has been visited
                firstPoint->setPath(connection, false);
                thisPoint = firstPoint->getPoint(connection);
                thisAngle = fmod(firstPoint->getAngles()->at(connection) +
                                 180, 360);

                firstAngle = firstPoint->getAngles()->at(connection);
                prevPoint = firstPoint;
                holePoints->push_back(firstPoint);

                // Stays iside of while loop during caluculations of one
                // hole
                while (true) {

                    // Loop trough the connections to find the previous
                    // point because it's angle is in intress.
                    for (int j = 0; j < thisPoint->getNumConnections(); j++)
                        {

                        if (prevPoint == thisPoint->getPoint(j)) {
```

```
                        pos = j;
                        break;
                    }
                }

                // Get the left most connection from which connection
                // comming from. Only allowed to use a connection that
                // is not allready tried or is set to be a dead end.
                for (int k = pos - 1; ; k--) {
                    if (k < 0){
                        k = thisPoint->getNumConnections()-1;
                    }
                    if (k == pos) {
                        checkedAll = true;
                        break;
                    }
                    else if (thisPoint->getPath(k) && thisPoint->
                             getEndFlag(k) == false) {

                        nextPoint = thisPoint->getPoint(k);
                        nextAngle = thisPoint->getAngles()->at(k);
                        thisPoint->setPath(k, false);
                        break;
                    }
                }

                // If all of the connections is already visited
                // break the while loop
                if (checkedAll) {
                    checkedAll = false;
                    break;
                }

                // Sum the inner angles of the polygon
                polygonAngle += fmod(360 + thisAngle - nextAngle,
                                     360.0f);

                // Save the point and go to the next one.
                holePoints->push_back(thisPoint);
                prevPoint = thisPoint;
                thisAngle = fmod(nextAngle + 180.0f, 360.0f);
                thisPoint = nextPoint;

                // If the hole is completed
                if (thisPoint == firstPoint) {
                    polygonAngle += fmod(360 + thisAngle - firstAngle,
                                         360.0f);
                    break;
                }
            }
        }

        // Check if the polygon was circumfered counter clockwise,
        // must be a bit wider limit because of some faults in
        // calculation with many decimals.
        if (polygonAngle > 179.9 * (holePoints->size() - 2) &&
            polygonAngle < 180.1 * (holePoints->size() - 2)) {

            holes->push_back(holePoints);
            holePoints = new vector<Point*>;
        }
        else {
            holePoints->clear();
        }
    }
  }
 }
}
```

```
/* Point.h - Data represenation class for the crack line segment end points.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#ifndef SPLITTER_POINT
#define SPLITTER_POINT

#include <vector>

using namespace std;

namespace splitter {
    class Point {
        public:
            /* Creates a Point object with unspecified coordinates.
             *
             * Input:  void
             * Output: A Point object.
             */
            Point();

            /* Creates a Point object with the specified coordinates.
             *
             * Input:  (x) the new X coordinate.
             *         (y) the new Y coordiante.
             * Output: A Point object.
             */
            Point(float x, float y);

            ~Point() { Point::numPoints--; }

            void setXY(float x, float y) { this->x = x; this->y = y; }

            void setX(float x) { this->x = x; }

            void setY(float y) { this->y = y; }

            /* Sets the intersection-test flag at position i to the specified
             * value. Note that the flag is shared between the two connected
             * points, so this method only has to be run on one of the points
             * for changes to take effect on both.
             *
             * Input:  (i) the position of the intersection-test flag to alter.
             *         (value) the new value of the test flag.
             * Output: void
             */
            void setTestFlag(short i, bool value);
```

```
   /* Sets the dead end flag at position i to the specified
 * value. Note that the flag is shared between the two connected
 * points, so this method only has to be run on one of the points
 * for changes to take effect on both.
 *
 * Input:  (i) the position of the dead end flag to alter.
 *         (value) the new value of dead end flag.
 * Output: void
 */
void setEndFlag(short i, bool value);

/* This metod is only used from the findHoles metod.
    * Sets the Path flag at position i to the specified value.
    * The path is available to use once from both directions
 * wich meens that every point has its own flag, for everey
    * connection. If the value is TRUE, the path is available to use
    * else (FALSE) the path is closed.
    *
    * Input: (i) the position of the flag to alter
    *        (value) the new value of the flag
    * Output: void
    */
void setPath(short i, bool value);

   const float getX() { return x; }

const float getY() { return y; }

/* Returns the specified intersection-test flag.
 *
 * Input:  (i) the position of the test flag of interest.
 * Output: TRUE if the crack section at position i has been tested
 *         before, FALSE otherwise. FALSE is also returned if
 *         there is no flag at position i. This happens only if
 *         i is larger than [number of points - 1].
 */
const bool getTestFlag(short i);

   /*
 * Returns the specified dead end flag.
 *
 * Input:  (i) the position of the dead end flag of interest.
 * Output: TRUE if the connection belongs to a dead end,
 *         FALSE otherwise.
 */
   const bool getEndFlag(short i);

   /* Returns the specified path flag.
 *
 * Input:  (i) the position of the path flag of interest.
 * Output: TRUE if the flag is available to use.
    *         FALSE if the connection is visited once before, from
    *         the same direction
 */
const bool getPath(short i);

   /* Returns the point’s unique ID
    *
    * Input: void
    * Output: The point’s ID
    */
   const int getID() { return ID; }


/* Retrieves the number of points connected to this point.
 *
 * Input:  void
 * Output: The number of neighbouring (connected) points.
 */
const short getNumConnections() { return angles->size(); }

/* Connects this point with the supplied point. The connection
 * is performed on both points. This method uses the addPoint()
 * method of both points.
```

```
             *
             * Input:  (p) Point object to connect to this point.
             * Output: void
             */
            void connect(Point* p);

            /* Disconnects the supplied point from this point, and vice versa.
             * This method uses the removePoint() method of both points. If the
             * supplied point and this point are not connected, nothing happens.
             *
             * Input:  (p) Point object to disconnect from this point.
             * Output: void
             */
            void disconnect(Point* p);

            /* Helper method for connect(). Adds the point p to this point.
             *
             * Input:  (p) Point object to add to this point.
             * Output: void
             */
            void addPoint(Point* p, float angle, bool* testFlag, bool* endFlag,
                          bool pathFlag);

            /* Helper method for disconnect(). Removes the point p from this
             * point. If p is not connected to this point, nothing happens.
             *
             * Input:  (p) Point object to remove from this point.
             * Output: void
             */
            void removePoint(Point* p);

            vector<Point*>* getPoints() { return points; }

            vector<float>* getAngles() { return angles; }

            /* Returns the connected Point object at position pos.
             *
             * Input:  (pos) The position of the Point object of interest.
             * Output: The connected Point object at the specified position.
             *         If there is no point at position pos, NULL is returned.
             *         This only happens if pos is larger than
             *         [number of points - 1].
             */
            Point* getPoint(short pos);

            static int numPoints; // The total number of existing Point objects.

        private:
            float x;
            float y;
            int ID;
            vector<Point*>* points;
            vector<float>* angles;
            vector<bool*>* testFlags;
            vector<bool*>* endFlags;
            vector<bool>* pathFlags;
    };
}

#endif
```

```
/* Point.cpp - Data represenation class for the crack line segment end points.
 *
 * Authors:
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include “Point.h”
#include <math.h>
#include <iostream>
#include <vector>

using namespace std;

namespace splitter {

    #ifndef SPLITTER_PI
    #define SPLITTER_PI
    const float PI = 3.14159265;
    #endif

    int Point::numPoints = 0;

    Point::Point() {
        ID = Point::numPoints++;
        points = new vector<Point*>;
        angles = new vector<float>;
        testFlags = new vector<bool*>;
        endFlags = new vector<bool*>;
        pathFlags = new vector<bool>;
    }

    Point::Point(float x, float y) {
        this->x = x;
        this->y = y;
        ID = Point::numPoints++;
        points = new vector<Point*>;
        angles = new vector<float>;
        testFlags = new vector<bool*>;
        endFlags = new vector<bool*>;
        pathFlags = new vector<bool>;
    }


    void Point::connect(Point* p) {
        // Calculate the angle between this and p
        float tempAngle = 0.0f;
        tempAngle = 180.0f / PI * atan2(p->getY() - getY(), p->getX() - getX());

        // Make sure tempAngle is in the interval [0, 360[
        tempAngle = fmod((tempAngle + 360.0f), 360.0f);
```

```
    bool* tempBool = new bool;
    *tempBool = false;

    bool* tempEnd = new bool;
    *tempEnd = false;

    bool tempPath = true;

    // Store the connection in the two points
    addPoint(p, tempAngle, tempBool, tempEnd, tempPath);
    p->addPoint(this, fmod((tempAngle + 180.0f), 360.0f), tempBool, tempEnd,
                tempPath);

}

void Point::addPoint(Point* p, float angle, bool* testFlag, bool* endFlag,
                     bool pathFlag) {

    // Find the correct position for the new value
    int insertPos = 0;
    for (; insertPos < angles->size() &&
           angles->at(insertPos) < angle; insertPos++) {;}

    // Push the rest of the angles (, points and flags) down a click
    if (angles->size() > 0) {
        angles->push_back(angles->at(angles->size() - 1));
        points->push_back(points->at(points->size() - 1));
        testFlags->push_back(testFlags->at(testFlags->size() - 1));
        endFlags->push_back(endFlags->at(endFlags->size() - 1));
        pathFlags->push_back(pathFlags->at(pathFlags->size() - 1));

        for (int i = angles->size() - 2; i > insertPos; i--) {
            angles->at(i) = angles->at(i - 1);
            points->at(i) = points->at(i - 1);
            testFlags->at(i) = testFlags->at(i - 1);
            endFlags->at(i) = endFlags->at(i - 1);
            pathFlags->at(i) = pathFlags->at(i - 1);
        }

        // Insert the new angle/point/flags
        angles->at(insertPos) = angle;
        points->at(insertPos) = p;
        testFlags->at(insertPos) = testFlag;
        endFlags->at(insertPos) = endFlag;
        pathFlags->at(insertPos) = pathFlag;


    } else {
        angles->push_back(angle);
        points->push_back(p);
        testFlags->push_back(testFlag);
        endFlags->push_back(endFlag);
        pathFlags->push_back(pathFlag);

    }


}

void Point::disconnect(Point* p) {
    removePoint(p);
    p->removePoint(this);
}

void Point::removePoint(Point* p) {
    for (int i = 0; i < points->size(); i++) {
        if (points->at(i) == p) {

            // Remove p from this, together with angle and flag.
            // Shift the rest of the points/angles/flags down properly.
            if (i < points->size()-1) {
                for (int j = i; j < points->size()-1; j++) {
                    points->at(j) = points->at(j+1);
                    angles->at(j) = angles->at(j+1);
```

```
                    testFlags->at(j) = testFlags->at(j+1);
                    endFlags->at(j) = endFlags->at(j+1);
                }
            }

            points->pop_back();
            angles->pop_back();
            testFlags->pop_back();
            endFlags->pop_back();
            break;
        }
    }
}

void Point::setEndFlag(short i, bool value) {
    if (i < endFlags->size()) {
        *(endFlags->at(i)) = value;
    }
}

const bool Point::getEndFlag(short i) {
    if (i < endFlags->size()) {
        return *(endFlags->at(i));
    } else {
        return false;
    }
}

void Point::setTestFlag(short i, bool value) {
    if (i < testFlags->size()) {
        *(testFlags->at(i)) = value;
    }
}

const bool Point::getTestFlag(short i) {
    if (i < testFlags->size()) {
        return *(testFlags->at(i));
    } else {
        return false;
    }
}
void Point::setPath(short i, bool value) {
    if (i < pathFlags->size()) {
        pathFlags->at(i) = value;
    }
}
const bool Point::getPath(short i) {
    if (i < pathFlags->size()) {
        return pathFlags->at(i);
    } else {
        return false;
    }
}

Point* Point::getPoint(short pos) {
    if (pos < points->size()) {
        return points->at(pos);
    } else {
        return NULL;
    }
}
}
```

```
/* room.h - File to create all primitives needed in graphics.h. Needs to include
 * Point.h to run. Is included in graphics.cpp.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <GL/glfw.h>
#include "Point.h"
using namespace std;

namespace splitter {

    /* Initialize all textures.
     *
     * Input:  void
     * Output: void
     */
    void initTextures();

    /* Draw the ceiling with a quad object and textures.
     *
     * Input:  void
     * Output: void
     */
    void drawCeiling();

    /* Draw the floor with a quad object and textures.
     *
     * Input:  void
     * Output: void
     */
    void drawFloor();

    /* Draw the surounding walls with a quad object and textures.
     *
     * Input:  void
     * Output: void
     */
    void drawWalls();

    /* Draw the middle wall with a hole for the glass window.
     * Applying textures for the block object.
     *
     * Input:  void
     * Output: void
     */
    void drawWindowFrame();

    /* Create a display list containing drawCeiling, drawFloor, drawWalls and
```

```
     * drawWindowFrame.
     *
     * Input:  void
     * Output: Index of the display list.
     */
    GLuint initRoomDisplayList();

    /* Draw the window with a quad object and textures.
     *
     * Input:  (holesVector) a vector containing the point objects defining the
     *         different holes created by the crack pattern.
     * Output: void
     */
    void drawWindow(vector<vector<Point*>* >* holesVector);

    /* Draw a block with size 2x by 2y by 2z, centered,
     * on the local origin, with texture coordinates for each face.
     *
     * Input:  (x) side is 2x
     *         (y) side is 2y
     *         (z) side is 2z
     * Output: void
     */
    void drawTexturedBlock(float x, float y, float z);

    /* Draw a quad with size 2x by 2y, centered,
     * on the local origin, with texture coordinates for each face.
     *
     * Input:  (x) side is 2x
     *         (y) side is 2y
     * Output: void
     */
    void drawTexturedQuad(float x, float y);
}
```

```
/* room.cpp - File to create all primitives needed in graphics.cpp. Includes
 * room.h, Point.h and tesselate.h.
 *
 * Authors:
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *         Jesper Carlsson,
 *         Daniel Enetoft,
 *         Anders Fjeldstad,
 *         Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <GL/glfw.h>
#include "room.h"
#include "Point.h"
#include "tesselate.h"

#ifndef SPLITTER_TEXTUREID
#define SPLITTER_TEXTUREID
//For textures, sets number of textures
GLuint textureID[5];
#endif
using namespace std;

namespace splitter {

     void initTextures() {
        // Load and set up textures - use GLFW to make this easier
        glEnable(GL_TEXTURE_2D); // Enable texturing
        glGenTextures(5, textureID); // Generate 5 unique texture IDs to use
        glBindTexture(GL_TEXTURE_2D, textureID[0]); // Activate first texture
        glfwLoadTexture2D("./textures/grycon.tga", GLFW_BUILD_MIPMAPS_BIT);
        glBindTexture(GL_TEXTURE_2D, textureID[1]); // Activate second texture
        glfwLoadTexture2D("./textures/yellobrk2.tga", GLFW_BUILD_MIPMAPS_BIT);
        glBindTexture(GL_TEXTURE_2D, textureID[2]); // Activate third texture
        glfwLoadTexture2D("./textures/yellobrk3.tga", GLFW_BUILD_MIPMAPS_BIT);
        glBindTexture(GL_TEXTURE_2D, textureID[3]); // Activate fourth texture
        glfwLoadTexture2D("./textures/yellobrk4.tga", GLFW_BUILD_MIPMAPS_BIT);
        glBindTexture(GL_TEXTURE_2D, textureID[4]); // Activate fifth texture
        glfwLoadTexture2D("./textures/stucco.tga", GLFW_BUILD_MIPMAPS_BIT);
    }

    GLuint initRoomDisplayList() {

        GLuint index = glGenLists(1);
        glNewList(1, GL_COMPILE);
            drawCeiling();
            drawFloor();
            drawWalls();
            drawWindowFrame();
        glEndList();

        return index;
    }
```

```
void drawCeiling() {
    //Draw the ceiling and set new texture
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, textureID[4]);
    glPushMatrix();
        glColor3f(1.0f, 1.0f, 1.0f);
        glTranslatef(0.0f, 10.0f, 0.0f);
        glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
        drawTexturedQuad(15.0f, 15.0f);
    glPopMatrix();
}

void drawFloor() {
    //Draw the floor
    glPushMatrix();
        glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
        drawTexturedQuad(15.0f, 15.0f);
    glPopMatrix();
}

void drawWalls() {
    //Draw walls and set new texture
    glBindTexture(GL_TEXTURE_2D, textureID[0]);
    glPushMatrix();
        glTranslatef(15.0f, 5.0f, 0.0f);
        glRotatef(-90.0f, 0.0f, 1.0f, 0.0f);
        drawTexturedQuad(15.0f, 5.0f);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(-15.0f, 5.0f, 0.0f);
        glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
        drawTexturedQuad(15.0f, 5.0f);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(0.0f, 5.0f, -15.0f);
        drawTexturedQuad(15.0f, 5.0f);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(0.0f, 5.0f, 15.0f);
        glRotatef(180.0f, 0.0f, 1.0f, 0.0f);
        drawTexturedQuad(15.0f, 5.0f);
    glPopMatrix();
}

void drawWindowFrame() {
    //Draw the window frame
    glBindTexture(GL_TEXTURE_2D, textureID[1]);
    glPushMatrix();
        glTranslatef(6.75f, 3.5f, 0.0f);
        drawTexturedBlock(3.25f, 3.5f, 0.25f);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(-6.75f, 3.5f, 0.0f);
        drawTexturedBlock(3.25f, 3.5f, 0.25f);
    glPopMatrix();

    glBindTexture(GL_TEXTURE_2D, textureID[2]);
    glPushMatrix();
        glTranslatef(9.25f, 8.5f, 0.0f);
        drawTexturedBlock(5.75f, 1.5f, 0.25f);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(-9.25f, 8.5f, 0.0f);
        drawTexturedBlock(5.75f, 1.5f, 0.25f);
    glPopMatrix();

    glBindTexture(GL_TEXTURE_2D, textureID[3]);
    glPushMatrix();
        glTranslatef(0.0f, 9.25f, 0.0f);
        drawTexturedBlock(3.5f, 0.75f, 0.25f);
```

```
    glPopMatrix();

    glPushMatrix();
        glTranslatef(0.0f, 0.75f, 0.0f);
        drawTexturedBlock(3.5f, 0.75f, 0.25f);
    glPopMatrix();
}

void drawWindow(vector<vector<Point*>* >* holesVector) {
    //Draw the glass window with blending and disable textures
    glEnable(GL_BLEND);
    glDisable(GL_TEXTURE_2D);
    glPushMatrix();
        glColor4f(1.0f, 1.0f, 1.0f, 0.5f); //need color with alpha channel

        if (holesVector->size() != 0) {
            tesselate(holesVector);
        } else {
            glTranslatef(0.0f, 5.0f, 0.0f);
            drawTexturedQuad(3.5f, 3.5f);
        }
    glPopMatrix();
    glEnable(GL_TEXTURE_2D);
    glDisable(GL_BLEND);
}

void drawTexturedBlock(float x, float y, float z) {

    glBegin(GL_QUADS);

        // Top face
        glNormal3f(0,1,0);
        glTexCoord2f(0.0f,1.0f);    glVertex3f(-x,y,z);
        glTexCoord2f(1.0f,1.0f);    glVertex3f(x,y,z);
        glTexCoord2f(1.0f,0.0f);    glVertex3f(x,y,-z);
        glTexCoord2f(0.0f,0.0f);    glVertex3f(-x,y,-z);

        // Back face
        glNormal3f(0,0,1);
        glTexCoord2f(0.0f,1.0f);    glVertex3f(-x,y,z);
        glTexCoord2f(0.0f,0.0f);    glVertex3f(-x,-y,z);
        glTexCoord2f(1.0f,0.0f);    glVertex3f(x,-y,z);
        glTexCoord2f(1.0f,1.0f);    glVertex3f(x,y,z);

        // Front face
        glNormal3f(0,0,-1);
        glTexCoord2f(0.0f,1.0f);    glVertex3f(-x,y,-z);
        glTexCoord2f(1.0f,1.0f);    glVertex3f(x,y,-z);
        glTexCoord2f(1.0f,0.0f);    glVertex3f(x,-y,-z);
        glTexCoord2f(0.0f,0.0f);    glVertex3f(-x,-y,-z);

        // Left face
        glNormal3f(1,0,0);
        glTexCoord2f(1.0f,0.0f);    glVertex3f(x,y,-z);
        glTexCoord2f(1.0f,1.0f);    glVertex3f(x,y,z);
        glTexCoord2f(0.0f,1.0f);    glVertex3f(x,-y,z);
        glTexCoord2f(0.0f,0.0f);    glVertex3f(x,-y,-z);

        // Right face
        glNormal3f(-1,0,0);
        glTexCoord2f(1.0f,1.0f);    glVertex3f(-x,y,z);
        glTexCoord2f(1.0f,0.0f);    glVertex3f(-x,y,-z);
        glTexCoord2f(0.0f,0.0f);    glVertex3f(-x,-y,-z);
        glTexCoord2f(0.0f,1.0f);    glVertex3f(-x,-y,z);

        // Bottom face
        glNormal3f(0,-1,0);
        glTexCoord2f(0.0f,1.0f);    glVertex3f(-x,-y,z);
        glTexCoord2f(0.0f,0.0f);    glVertex3f(-x,-y,-z);
        glTexCoord2f(1.0f,0.0f);    glVertex3f(x,-y,-z);
        glTexCoord2f(1.0f,1.0f);    glVertex3f(x,-y,z);
    glEnd();
}

void drawTexturedQuad(float x, float y) {
```

```
        glBegin(GL_QUADS);
            // Front face
            glNormal3f(0,0,1);
            glTexCoord2f(0.0f,1.0f);    glVertex3f(-x,y,0.0f);
            glTexCoord2f(0.0f,0.0f);    glVertex3f(-x,-y,0.0f);
            glTexCoord2f(1.0f,0.0f);    glVertex3f(x,-y,0.0f);
            glTexCoord2f(1.0f,1.0f);    glVertex3f(x,y,0.0f);
        glEnd();
    }
}
```

```
/* tesselate.h -    Contains functions that tesselates the glass
 *                  according to the holes.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <vector>
#include <GL/glfw.h>
#include "Point.h"
using namespace std;
using namespace splitter;

/* Resets the objektlist containing the tesselated window
 */
void resetTessList();

/* Tessellates the window with holes into renderable polygons.
 * A display list is used since the holes only needs to be tesselated once.
 *
 * Input:  (tessVector) A pointer to a vector with holes, each hole is
 *         pointer to another vector containing the points that is the hole.
 *         The points is sorted counter clockwise.
 * Output: void
 */
void tesselate(vector<vector<Point*>* >* tessVector);

/* Callback functions that is run at appropiate times during the
 * tessellation
 */
void CALLBACK beginCallback(GLenum which);
void CALLBACK errorCallback(GLenum errorCode);
void CALLBACK endCallback(void);
void CALLBACK vertexCallback(GLvoid *vertex);
void CALLBACK combineCallback(GLdouble coords[3],
                              GLdouble *vertex_data[4],
                              GLfloat weight[4], GLdouble **dataOut);
```

```
/* tesselate.cpp -  Contains functions that tesselates the glass
 *                  according to the holes.
 *
 * Authors:
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * Copyright 2004,
 *          Jesper Carlsson,
 *          Daniel Enetoft,
 *          Anders Fjeldstad,
 *          Kristofer Gärdeborg.
 *
 * This file is part of Splitter.
 *
 * Splitter is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Splitter is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Splitter; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <vector>
#include “Point.h”
#include “tesselate.h”
#include <GL/glfw.h>

using namespace std;

static GLuint objektList;

void resetTessList() {
    glDeleteLists(objektList, 1);
}

//The callback functions is called during tessellation at appropriate times
void CALLBACK beginCallback(GLenum which) {
    glBegin(which);
}

void CALLBACK errorCallback(GLenum errorCode) {
    const GLubyte *estring;

    estring = gluErrorString(errorCode);
    fprintf(stderr, “Tessellation Error: %s\n”, estring);
    exit(0);
}

void CALLBACK endCallback(void) {
    glEnd();
}

void CALLBACK vertexCallback(GLvoid *vertex) {
    const GLdouble *pointer;
    glVertex3dv((GLdouble*)vertex);
}

void CALLBACK combineCallback(GLdouble coords[3],
                                GLdouble *vertex_data[4],
                                GLfloat weight[4],
                           GLdouble **dataOut ) {

    GLdouble *vertex;
    int i;
```

```
    vertex = (GLdouble *) malloc(6 * sizeof(GLdouble));

    vertex[0] = coords[0];
    vertex[1] = coords[1];
    vertex[2] = coords[2];
    *dataOut = vertex;
}


void tesselate(vector<vector<Point*>* >* tessVector) {

     if(!glIsList(objektList)) {

        objektList = glGenLists(1);
        GLUtesselator *tobj;
        tobj = gluNewTess();

        //Defines the callback functions
        gluTessCallback(tobj, GLU_TESS_VERTEX,
                        (void (CALLBACK*)())vertexCallback);
        gluTessCallback(tobj, GLU_TESS_BEGIN,
                        (void (CALLBACK*)())beginCallback);
        gluTessCallback(tobj, GLU_TESS_END,
                        (void (CALLBACK*)())endCallback);
        gluTessCallback(tobj, GLU_TESS_ERROR,
                        (void (CALLBACK*)())errorCallback);
        gluTessCallback(tobj, GLU_TESS_COMBINE,
                        (void (CALLBACK*)())combineCallback);

        vector<GLdouble**>* vertexVector = new vector<GLdouble**>;

        // Stores the window corners in the GLdouble array since
        // that is required by gluTexVertex
        GLdouble** vertexList = new GLdouble*[4];

        vertexList[0] = new GLdouble[3];
        vertexList[0][0] = -3.5;
        vertexList[0][1] = 8.5;
        vertexList[0][2] = 0.0;
        vertexList[1] = new GLdouble[3];
        vertexList[1][0] = 3.5;
        vertexList[1][1] = 8.5;
        vertexList[1][2] = 0.0;
        vertexList[2] = new GLdouble[3];
        vertexList[2][0] = 3.5;
        vertexList[2][1] = 1.5;
        vertexList[2][2] = 0.0;
        vertexList[3] = new GLdouble[3];
        vertexList[3][0] = -3.5;
        vertexList[3][1] = 1.5;
        vertexList[3][2] = 0.0;

        vertexVector->push_back(vertexList);

        //Stores all the holes in GLdouble arrays
        for(int i = 0; i<tessVector->size(); i++) {

            //Creates a double array with the same number of rows as
            //the size of the tessVector
            GLdouble** vertexList =
                new GLdouble*[tessVector->at(i)->size()];

            for(int j = 0; j<tessVector->at(i)->size() ; j++) {
                vertexList[j] = new GLdouble[3];
                vertexList[j][0] = tessVector->at(i)->at(j)->getX();
                vertexList[j][1] = tessVector->at(i)->at(j)->getY();
                vertexList[j][2] = 0.0;
            }
            vertexVector->push_back(vertexList);
        }

        glNewList(objektList, GL_COMPILE_AND_EXECUTE);
        gluTessBeginPolygon(tobj, NULL);

        //Using positive rule so since the holes is sorted in counter
```

```
        //clockwise order. The glass is defined clockwise
        gluTessProperty(tobj, GLU_TESS_WINDING_RULE,
                        GLU_TESS_WINDING_POSITIVE);

        //Draws the window
        gluTessBeginContour(tobj);
        for (int i = 3; i >= 0; i--) {
            gluTessVertex(tobj,vertexVector->at(0)[i],
                          vertexVector->at(0)[i]);
        }
        gluTessEndContour(tobj);

        //Draws the contour of the holes, starts from behind since the
        //holes is defined counter clockwise
        for(int i = 1 ; i < vertexVector->size(); i++) {
            gluTessBeginContour(tobj);

            for (int j = tessVector->at(i-1)->size()-1 ; j >= 0 ; j--) {
                gluTessVertex(tobj,vertexVector->at(i)[j],
                              vertexVector->at(i)[j]);
            }

            gluTessEndContour(tobj);
        }

        gluTessEndPolygon(tobj);
        glEndList();
        gluDeleteTess(tobj);
    }
    else {
        glCallList(objektList);
    }
}
```