A Real-Time application to visualize the results of a Branch & Bound algorithm

Project for the Real-Time Graphics Programming course held by Professor Davide Gadia, University of Milan, November 2023

Alessio La Greca

Abstract

In the wide field of optimization problems, a rich subset is that of the variants of the Knapsack problem. In particular, one of these variants caught our attention: that of the two-dimensional packing problem. In this version, the knapsack is defined by a width W and a height H, and the items to be put in the knapsack too have a width and a height. Under the (*not-so*) simplifying assumption that the value of each item is equal to its area, the objective is to find the best placement of the items, from now on, called "boxes", in the knapsack, also called "container". The best possible placement is the one that maximizes the total profit of the boxes inserted, or equivalently, that minimizes the wasted, unoccupied, area. To tackle this problem, a Branch & Bound algorithm, that performs an implicit enumeration of the possible solutions, has been developed. However, our focus won't be on the algorithm itself, but on a problem that came up after: the difficulty in interpreting the results of a run. How can we efficiently take a node of the B&B algorithm, which represents a (partial) placement of boxes in the container, and understand immediately what it means? After all, we are not only interested in numerical data such as the Primal or Dual bound of that node, which are easy to read and understand, but also in the actual way the boxes were placed. After all, while the data structures and formats used by the algorithm might allow it to work efficiently, they are not very readable, from a human's perspective. The ability to interpret in a fast and clear way the algorithm's solutions is not to be underestimated: it can allow the programmer to understand if the algorithm worked as intended, compare multiple executions with different parameters, and also, why not, help him come up with new ideas to improve the B&B procedure. These are the main reasons behind the development of this project: having an application capable of reading the output data of a B&B run and display them clearly to the user. To to this, we leveraged on OpenGL, focusing on two levels of detail: node-specific and run-specific.

Contents

1 Introduction

The optimization problem tackled is a variation of the original Knapsack problem, namely, the two-dimensional packing problem. Actually, the literature seems to be a bit confused regarding the standard way to call it. This is because even the slightest variation of the formulation can change the problem. In short, however, the problem is: given a two-dimensional container of width W and height H, and a set of boxes, each with its own width and height, we want to find the best possible packing for these boxes in the container, that is, the packing that maximises the occupied area (or equivalently, that minimizes the unoccupied area). This problem finds many real-world applications, such as the cutting of rectangular sub-surfaces from a master surface made of cloth to produce clothes. That said, our objective is not to give an exhaustive definition of the problem, but rather the main characteristics that define the problem as well as its solution space. Stating them will be useful to understand our implementation choices. Such characteristics are:

- *Orthogonality*: the container and the boxes are rectangles, and while it is true that we could, in principle, place a box in any given and feasible rotation, we assume that a box is always placed in such a way that its edges are orthogonal to the container's edges. Whether we allow the input boxes to rotate 90 degrees or not, depends on the specific instance considered.
- *Profits proportional to area*: a given box of width w_i and height h_i has a profit of $w_i * h_i$. This is an arbitrary choice, as in a more general case, each box is allowed to have a positive profit value.
- *Pre-placed obstacles*: we allow the container to have, wherever we prefer, obstacles, that can be considered of zero-profit boxes with a given width and height. In real applications, this can be considered as having a defect on the master surface that can't be considered when performing the cutting of such surface. Since it would be too complex to model each defect shape, we limit ourselves to defects in the shape of rectangles orthogonal to the master surface.
- *Packing*: a general two-dimensional knapsack problem can be tackled either with a cutting or a packing approach. The difference is the following:
  - In the *cutting* case, we are only allowed to perform guillotine cuts to the container, that is, cut in half the container with a horizontal or vertical cut, and then do this recursively on the two sub-rectangles created. This recursion stops once a certain condition has been met, and then to each sub-rectangle obtained in this way is associated a box that can feasibly stay in that region. The box is then cut from the corresponding region.
  - The *packing* case is instead the most general one, because we allow also patterns that are not obtainable through just guillotine cuts. This expands a lot the solution space, which both increases the complexity of the exploration as well as the value of the optimal solution that can be found.

Our B&B algorithm has been developed for the packing case.

- *Unconstrained*: given a box of width w_i and h_i, we might have one or more copies of it. Although it is true that we can treat boxes with the same width and height as completely different boxes, we found more efficient and effective the definition of *box types* and their *quantity*. This means that, if our problem has n input boxes with the same width and height, we create one new box type, uniquely identified by their w_i and h_i, setting its amout of available boxes to, initially, n. This means that we have n available copies of the same box type, with dimensions w_i and h_i, to be placed.

Given these assumptions, we can start talking more in-depth about our problem: being able to visualize in a meaningful way a solution of the Branch & Bound algorithm.

2 Representation of solutions

2.1 Format of a B&B node

When dealing with an optimization problem, we can always perform an exhaustive search to find its optimal solution. However, in practice, this is never done, since the search space is usually so vast that an exhaustive approach would require way too much time to give an output. So, what we can do, is either use an heuristic and stick with a possibly sub-optimal solution, or use an implicit enumeration approach. Such approaches are guaranteed to find the optimal solution, as in the exhaustive one, but run faster by cutting off parts of the solution space that are demonstrated to be sub-optimal. Algorithms that do this kind of search use either Dynamic Programming or Branch & Bound techniques, and the latter is our case. Without going too much in the details, our Branch & Bound algorithm works as follows:

1. Start with an empty solution, that in our case, is the empty container. This will be the initial node. Please note that usually when we talk about a solution, we talk about all the feasible ones, so yes, even deciding not to put any box in the container is a solution (although one that is probably sub-optimal). Moreover, **we will use the terms "node" and "solution" as synonyms**. This is because a B&B algorithm reasons in terms of nodes, where a node usually encapsulates a specific solution and some additional informations, such as the *Primal Bound* and the *Dual Bound* of that solution. From each node, zero, one, or multiple new nodes can stem. Since in our case a node is basically represented as a partial placement of boxes in the container, where adding a box to this partial placement leads to a new partial placement and so a new and different solution, the two terms can be considered as the same. Declare a global variable, like *best_PB*, and set it initially to 0. Put also the initial node in a queue (or a list, it really depends on the implementation).

2. While the queue is not empty, repeat:

   (a) Get the head of the queue, and compute its Primal Bound (PB), that is, its value according to the objective function. In our case, it's the same as computing the total area of the boxes placed. If PB $>$ *best_PB*, *best_PB* $=$ PB.

   (b) Compute the Dual Bound (DB) of this node. A Dual Bound is an overestimate of the optimal value that we can get by starting with the current partial solution. A standard (but not the only) way to compute it is to relax some constraints of the original problem, so that we get an easier problem to solve, in terms of time spent. For example, given the boxes that are still to be placed and the remaining free space in the container, we could solve a classic Knapsack problem. Its value is an overestimate, because probably, because of the geometric constraints of the original problem, not all the boxes belonging to the optimal solution of the Knapsack problem can actually be placed in the container starting from this partial solution. However, the important thing is that this value is $\geq$ than the actual optimal value we can get starting from this partial placement.

   (c) If $DB < best\_PB$, this node can be closed, because we have the guarantee that no solution created from the current one can be the optimum. The node is also closed if it is not possible to place any remaining box in any empty position of the current, partially occupied, container.

   (d) Otherwise, choose a position in the container. For each remaining box that can be feasibly placed in that position, generate a new node, equal to the current one, but with the addition of the new box in that position. Each of the new nodes is then added to the queue. The queue can of course be sorted in some way according to a specific *search* *strategy*, granting for example that the next node explored is always the one with the highest PB, therefore following a greedy approach (anyway, this is just an example: in our algorithm, we implemented over fourty different search strategies, with the aim of finding the optimum as soon as possible).

2.2 Additional constraints and features

There are some more things to take into account when dealing with a node of our B&B procedure:

- The width W and the height H of the container, as well as those of an arbitrary box or obstacle, are positive integers. This means that, when choosing a pair of coordinates (x, y) on which to place a box, these always belong to the range $[0, W-1]$ and $[0, H-1]$, and can only be integer.
- It is important to define what it means to place a box at the coordinates (x, y). A rectangular box has, by definition, four corners: the top-left, the top-right, the bottom-left and the bottom-right. Assuming that box i has width w_i and height h_i, placing a box at the coordinates (x, y) means to place the bottom-left corner of that box there, therefore fully occupying the space defined by the set of coordinates given by $[x, x+w_i) \times [y, y+h_i)$.
- When choosing, from the current node, the coordinates in which the new box will be placed, not all coordinates can be considered. Of course, those corresponding to a region of the container occupied by a box or an obstacle are not taken into account. However, among all the coordinates corresponding to an empty region of space, only those that have an edge on the left and one below are considered to be available. For example, in the root node (the empty container), a box can only be placed on the bottom-left corner of the container. If a box has been placed there, the next one will need to be either placed on its right, bottom-justified, or above it, left-justified, and so on.
- Deciding to place boxes in a certain way can imply that some regions of the container become unavailable. Assume that, in a given instance, the smallest box has dimensions w_{min} and h_{min}, the biggest has dimensions w_{max} and h_{max}, and the container is $W \times H$. If the biggest box is placed first and $(w_{min} + w_{max} > W) \land (h_{min} + h_{max} > H)$, then no box can be placed on the right or above the one placed. It can be useful to determine, geometrically, the space that, no matter what, can no longer be occupied in this partial placement. Since this kind of space can always be decomposed in a set of rectangluar shapes, we define a new concept: that of the *false boxes*. A false box is the same as an obstacle, because while it has a width and a height, its profit value is 0. The only difference is that, while the obstacles are given as inputs, the false boxes are placed arbitrarily by the algorithm in those regions of the current partially filled container that can't possibly be filled with a normal box. This helps strengthen the Dual Bound. Here we make a distinction: while the term *false boxes* identifies zero-value boxes placed by the algorithm to flag a wasted area of the current solution, the term *true boxes* refers to the input boxes.
- Our placement of the boxes is bottom-left justified, as mentioned previously. Without going too much into details, our implementation not only stores the placements of the true and false boxes, but also the *projections* of the vertices (literally, any corner of a box) on the left and below. This information is important to be kept, since it allows to distinguish nodes in which a certain preprocessing can be performed or not, as well as granting other speed-ups.

2.3 Input file data format

Now that we have described what is a node and what are its main features, we can state the format of the input file received by the application, that will be used to decide what needs to be seen.
First of all, two files are given as input:

- *nodesNumber.txt*: it contains only one line, a positive integer representing the number of B&B nodes explored during the run of the algorithm.
- *nodesInformations.txt*: it's the main input file of the application. It contains all the informations regarding the instance considered and the run performed: the container dimensions, the obstacles, and all the nodes explored.

What follows is a description of the input file *nodesInformations.txt*, given line-by-line, useful to interpret its text lines:

1. The width W and height H of the container;
2. A dividing line "- - - - -". We will see this many times. It's mainly used to tell to the function that reads this input file that the current set of informations is finished.
3. A list of the obstacles present in the container, each identified by a line of the kind $x0_i$ $y0_i$ w_i h_i. The first two values identify the bottom-left coordinates of the obstacle, while the last two tell its width and height;
4. A dividing line "- - - - -";

Then, each node, ranging from 0 to $n-1$, where n is the positive integer value red from *nodesNumber.txt*, is described. The format of a node is as follows, once again, line-by-line:

1. The *exploration ID* of the node. It identifies the moment the node was explored;
2. The *father ID* of the node. It identifies the exploration ID of the father node of this node, that is, the node that, after branching, created the current node;
3. The *creation ID* of the node. It identifies the moment the node was created. Please note that it is not the same as the exploration ID: a child node of the root node has a low creation ID, because it's one of the first ever created, but it might be explored much later on during the search;
4. The *PB*, or *Primal Bound*, of the node. It represents the sum of the areas of the true boxes placed;
5. The *DB*, or *Dual Bound*, of the node. It represents the overestimate of the optimal value that we can get starting from this node. If equal to -1, it means that the current node was closed due to preprocessing;
6. The *best PB*. It's the value of the best solution found up until now;
7. The *level in tree*. It represents the depth of the node in the B&B tree built. The root node will have level = 0, its children 1, and so on;
8. A list of the remaining quantities for each box type. In our algorithm, every box is identified by an unique integer, ranging from 0 to $k-1$, if we have k different types of boxes. So, this will be a list of k positive integers, where the $i-th$ one represents the number of copies of the $i-th$ type of box that can still be placed starting from the current node's solution;
9. A list of the true boxes placed in the container, each identified by a line of the kind $x0_i$ $y0_i$ w_i h_i ID_i. The first four parameters are the same as the obstacles', while the last one is the unique ID of that box type.
10. A dividing line "- - - - -";
11. A list of the false boxes placed in the container, each identified by a line of the kind $x0_i$ $y0_i$ w_i h_i.
12. A dividing line "- - - - -";
13. A list of the projections present in the container, each identified by a line of the kind $x0_i$ $y0_i$ $x1_i$ $y1_i$. In this case, we have a set of two points, that identify the extremes of the projection. It follows that either the two x-coordinates or the two y-coordinates will be the same.
14. One last dividing line "- - - - -";

3 Node Visualizer Application

The first part of the project consists in an OpenGL application capable of reading the input files, create a representation in memory for each node, and finally render in a human-readable way a given node. The user will then be able to switch the current node, to decide which node is worth visualizing.

3.1 Data structures

When modeling a node, there are three main informations we want to be able to represent: the true boxes placed, the false boxes placed, and the projections present. To do that, three classes were created:

- *Box.cpp*: it contains the bottom-left coordinates of a box, its dimensions and its ID.
- *Projection.cpp*: it contains the two two-dimensional coordinates of a projection.
- *TreeNode.cpp*: it contains all the informations regarding a node and the solution it represents: its exploration, father and creation IDs, the PB, the DB, the best PB found up until that moment, and the level of the node in the tree. It also contains three arrays: one for the true boxes placed, one one for the false boxes placed, and one for the projections present. The lengths of these arrays are stored too.

At the beginning of the execution, the input files are read:

- We store in two integer variables, *wContainer* and *hContainer*, the dimensions of the container;
- We store in a Box array the obstacles;
- We store in a TreeNode array the nodes explored in the run.

We also create an integer variable, *currentNodeIndex*, initially set to 0, that will represent the node we are visualizing.

3.2 Node representation

We want all the informations of a node to be displayed on screen in an understandable way. These informations can be gathered in three main sets:

1. The container, its boxes and its projections. These informations will be on the lower-left side of the screen;
2. The various IDs, the Bounds and the level in tree. These informations will be on the upper-left side of the screen;
3. The remaining quantities of each box type and the container dimensions. These informations will be on the right side of the screen.

Let's see how we approached the representation of each of them individually.

3.2.1 Primitives rendering approach

Usually, when reasoning about Real-Time rendering, we think about three-dimensional models placed on a three-dimensional world. It is after the a series of transforms application that we bring them in two dimensions. But in our case, what we actually want to render are two-dimensional, static informations: just a bunch of rectangles and texts on screen (that can be reduced to quads, or rectangles, with a texture applied to them). Therefore, a dilemma occours: should we render two-dimensional objects in a three-dimensional world and then perform an orthographic projection, or reason directly in terms of Normalized Device Coordinates, since everything is two-dimensional? We don't think there is a right answer. The first approach has the advantage that is more standard and probably easier to apply, even if it has the overhead of performing a bunch of transforms at every frame that in practice are not necessary (even though there are many ways to speed-up such process). The second, instead, gives more customizability if things are handled with care, although it requres much more attention programmer-wise. We personally decided to go with the second, both because we like the idea of high customizability, and because, reasoning in terms of NDC, we had the opportunity to reason at a low level regarding the placement of the primitives.

3.2.2 Container, boxes and projections

The current solution, represented by the container and its contents, will be shown in the lower-left part of the screen. The very first problem we came up with when deciding to represent the container was: how much space of the window should be dedicated to it? It shouldn't be too little, otherwise we wouldn't understand the current solution, but not too big either, or else there wouldn't be much space for the other informations. For this reason, a variable between 0 and 1, *maxPortionDedicatedToContainer*, decides the percentage of the window that we should give to the container. Since here we always reason in NDC, this means that all the space in $[0, 0.8 * window\_width) \times [0, 0.8 * window\_height)$ is dedicated to the container. However, we don't want to stretch the container and its boxes to match this space. If the container is tall and thin, its proportions should be mantained in the best possible way. Same if it is long and short. Not only: say the container is a square (the width is the same as the height). In such a case, one would think that such container should fill the entire portion of the screen dedicated to it. But if the window is not squared, but is, for example, 1920 x 1080, the container and the boxes would be stretched horizontally. Since we want to be as faithful as possible to the input dimensions (otherwise, we would betray the idea of having a visualization application that gives a human-readable output of each partial solution), we first do a bit of preprocessing, and rescale the width and the height of the container so that it matches the aspect ratio of the window. This is necessary, since in the end, we will have to manually specify the x and y coordinates of the primitives we want to draw. Once this is done, we have the coordinates of both the lower-left and upper-right corners of the container. This is useful when rendering the boxes and the obstacles inside of it.
Then, the rendering of the boxes happen. The boxes to render, both true and false ones, are taken from the TreeNode corresponding to the node we want to render. For their rendering, we created a VAO, a VBO and an EBO. The idea is to have four vertices (two triangles sharing two vertices), each of which has three-dimensional coordinates (although the z-coordinate is not important) and an rgb value for the color of the box. The coordinates of the four vertices are taken from the box (x_0, y_0) coordinates, as well as from its width w_i and height h_i. With these data, it is possible to determine a box's lower-left and upper-right coordinates, from which we can obtain the remaining two corners. All of these coordinates are expressed as Normalized Device Coordinates.
For the obstacles, it's simple: just take the four corner coordinates and convert them in vertices to put in the VBO. Note that we use just one VAO for the rendering of a quad, updating the contents of the VBO with *glBufferSubData* each time a new box has to be rendered. The color of an obstacle is always the same of the container (we used black).
For the false boxes, it's basically the same as the obstacles. The only difference is in the color, white, in this case.
The true boxes require a bit more care. The first problem is their color: which color should a box have? Since we want the output to be understandable, it suffices that every box has a color different from the others. Therefore, one idea could be to generate a new color randomly every time a box is drawn and hope it's not the same as one previously generated. But if two or more copies of the same box type are placed in the same solution, it would make sense to have them of the same color. In order to achieve this degree of consistency and make sure no color is repeated, we implemented a function, *getColorFromID* (found in the *utils.cpp* file), that takes a positive integer as input and returns, in a deterministic way, a color generated from that number. In this way, given the ID of a box type, an unique color can be generated and associated to it. While this is useful to distinguish different kind of boxes, it doesn't help in understanding the nature of a specific box. In particular, we would like to know, for each placed box, its ID and its dimensions. To do this, we leveraged on the TextRendering section of learnOpenGL, and developed the *RenderText* function. In this way, we just have to specify the coordinates of the text, a scaling factor for its font size, a color, and of course the text itself that we want to render. It's not a very efficient function, as each letter drawn requires to draw a quad on top of which is applied a texture, but this is not a problem for now, since the number of polygons we want to render is very limited, and most importantly, a node's graphic representation is always static (nothing moves). With this function, it's possible to render, on top of a box, both its ID as well as its width and height.
Not much needs to be said about the vertex and fragment shaders that render the quads (*shader_node_info*). The vertex shader simply takes a vertex, whose attributes are just its position and its color, and places the vertex at the given position, passing the color to the fragment shader. This last shader simply returns this color in the output vec4 *FragColor*, without further processing.

3.2.3 IDs, Bounds and level in tree

The Exploration, Father and Creation IDs, the Primal, Dual and Best Primal bounds, and the level in the B&B tree of the current node are shown in the upper-left corner of the screen. To draw them, we leverage again on the *RenderText* function, supplying to it the coordinates at which each text must be drawn. The numeric data do be shown are actually red from the TreeNode object representing the current node.

3.2.4 Remaining quantities and container dimensions

On the right side of the window, we show the remaining number of copies of each box type. To make things more clear, we drew behind every box type ID a small quad of the color associated to that ID. On their right, the number of copies remaining is shown. On the very bottom-right side of the window, the container dimensions are shown. The procedures and shaders used to render these informations are the same as the container and boxes case.

3.2.5 Input handling

As stated, this application can read a file containing the description of more than one partial solution. Only one node is shown during a certain frame though. Specifically, once the TreeNode array has been filled with n TreeNode objects, one for each node, the variable *currentNodeIndex* decides which node should be shown during the current frame. Since the nodes are stored in the array in increasing order of Exploration ID, *currentNodeIndex* effectively represents the Exploration ID of the node we want to render. The variable is initially 0, so the first node shown is the root node of the B&B tree: the empty container. By changing the value of this variable, we can actually decide the node that must be rendered at the current frame. This is done using the arrow keys.
We first define a variable, *nodesToAdvance*, and set it to 1. The reason we need this variable will be clear later on. When the right key is pressed, *currentNodeIndex* is incremented by *nodesToAdvance*, effectively bringing us to the node explored right after the current one, and that will be rendered in the next frame. With the left key a subtraction occours instead of an increment (*currentNodeIndex* is always clamped between 0 and $n-1$). To avoid breaking a finger by mashing an arrow key when we wish to go many nodes ahead or behind, it's possible to keep shift pressed. In this way, keeping the right or left key pressed will quickly repeat the command of moving to the next or previous node, as long as the arrow key is kept pressed. That said, there are cases, such as when we deal with trees with thousands of nodes, in which this procedure is still too slow to allow the user to efficiently move from a node to another, maybe close to the end of the TreeNode array. This is why, by pressing "." (dot) and "," (comma) keys, one can multiply or divide by 10 the value of *nodesToAdvance*, so that, when using the arrow keys, the number of nodes "jumped" forward or backward is much greater. To give a feedback to the user, we also render, when the dot or comma key is pressed, the new value of *nodesToAdvance*, that fades out in a bunch of seconds. The rendering and fading of such text is handled by a function, *drawSpeed*, found in the *Drawing.cpp* file.

4 Tree Visualizer Application

Being able to see in an understandable way each partial solution and quickly move through them is incredibly helpful, since it allows to effortlessly see many boxes configurations and reason about them, what brought the algorithm there and how it "reasoned" immediatly after or before. However, it might not be enough. With the Node Visualizer Application, only informations related to a specific node can be accessed, but it's difficult to tell the "weight" each node has had in the search.
What we mean is that the efficiency of a B&B algorithm is given, among other things, primarily by the number of nodes explored. The less they are, the better it is. Therefore, if a sub-optimal node has had many children, that node is probably worth studying, with the hope to find a way to change the algorithm and avoid to explore it in the next run. But with what we have now, it's difficult to tell if a node had many children. Of course we could compute this number ourselves, or let the algorithm do it for us and show it among the IDs, Bounds and level. But still, it wouldn't be the full picture. Do we want the total number of nodes belonging to the the sub-tree rooted in a specific node? Or just the direct children nodes count? Or both? But even with this information, wouldn't it be useful to understand if the search went deep in the direction of that node, exploring its children first and the rest later, or the opposite? All these variable informations that

we thought could be interesting to analyze led to the developing of the second part of this project: an OpenGL application capable of rendering the B&B tree... in its entirety! After all, what the algorithm explores is an actual tree, where each node generates (potentially) many others.

4.1 The final goal

In representing the tree, we don't want to do anything fancy: just a clear representation of the nodes and their parenthood relationship is enough. Therefore, our implementation objectives are:

- Having a square for each node explored;
- Writing, on top of each square, its Exploration ID;
- Having lines connecting each node to its children. We will call these elements "*bridges*";
- If possible, give a different color to the optimal node, so that it immediately stands out;
- Since, after having identified a possibly problematic node, we would also like to see it, we also want to be able to click on a node with the mouse cursor and visualize its content, by using the same procedures seen in the previous chapter. *To exit a node and go back to see the entire tree, we will use the Spacebar key.*
- Since we are talking, most of the time, of trees that have a very high number of nodes, it would be impractical to fit the whole tree on a given window. We could do that, but the nodes would be so little that we wouldn't be able to read their Exploration ID, nor understand their parenthood relationships. Therefore, we actually want the user to have the ability to control the way the tree is seen. This translates to being able to zoom in and out (so that one can either see few nodes at a given time, but they are all clearly visible, or have many on screen to get a grasp of the overall tree structure, while the single nodes are less readable) and move around the scene to check other parts of the tree.

We decided to represent the tree in two dimensions, and not three, so to have it be more clear. That said, we will render the nodes and the lines in a three-dimensional space to allow horizontal and vertical movement as well as the ability to zoom in and out. Please note that, with the term "node", here we refer to a squared quad placed in the scene, representing a node of the B&B tree. By clicking on it, we will be able to "open" it to see its solution.

4.2 Positioning the nodes

Rendering a quad is not a problem for us. After all, it's just two triangles sharing two vertices. We just have to create ourselves a unit quad, whose vertices will be located initially at coordinates $(-0.5, -0.5)$, $(0.5, -0.5)$, $(-0.5, 0.5)$ and $(0.5, 0.5)$, and then translate it wherever we want with a transform matrix, that will only apply a translation. The true problem is deciding how and where to place each node, so that it will then be clear who is the father of who and at which level of the tree a node is located. To do that, we decided to create an array, *nodesPositions*, that will store as vec3 the coordinates of the central point of every quad representing a node. In this way we will be able to build the corresponding translation matrix for each node and that will be it. To do that, we call the *addModelMatrix_Nodes* function (located in the *Drawing.cpp* file) passing as argument the empty *nodesPositions* array, an additional array of colors of the same length, and the TreeNode vector containing the informations regarding every node. The key idea of this function is to recursively determine the placement of every node, based both on local and global informations. It works as follows:

1. If this is the first time the function is called, Initialize a global vec3 variable, *coordinates-NewNodeToDraw*, to $(0, 0, -100)$ (we will mention the Z-value of every vec3, though we won't dwell much on them: it suffices to know that they are set in such a way that all the informations rendered have the correct depth, i.e. all the text rendered will be on top of everything). It will represent the coordinates at which the current node must be rendered.
2. Change the y component of the *coordinatesNewNodeToDraw* so that it is proportional to the level in the tree of the node. Then save on the *nodesPositions* array a copy of this vec3, and save also on the corresponding TreeNode these coordinates.
3. Save on the color array the color that the current node will have. White for all nodes but the optimum one, green for the latter.

4. If the current node has some children nodes, loop over them.
    (a) If we are dealing with the very first children, just call recursively the function.
    (b) Else, add some horizontal scalar offset to the x component of *coordinatesNewNodeToDraw* and then call the function recursively.

    This allows to give horizontal spacing between nodes when it is necessary. The vertical spacing instead is implicitly handled by the level in tree of each node.

At the end of the procedure, we have the complete list of vec3 positions that each node should have. They are sorted in a Depth First fashion however, but that's not a problem, since we will take into account this feature when we use them.

4.3 Positioning the bridges

After we have decided where the nodes should be placed, it's time to create the bridges that connect them, so that the parenthood relationships are evident. To keep things simple,the bridges will just be horizontal and vertical lines, that together will create connections (*bridges*) between nodes.
These lines however should be somewhat thick, otherwise they wouldn't be much visible. That's why we decided to use rectangular quads for them. However, creating the rectangles we need beforehand and then use them when needed doesn't seem a very good choice. In general the size of these lines depends on the nodes to connect, therefore there would be little to no room for customization if we had to query a certain rectangle from a "standard" set of quads. That's why a more robust and modular approach was chosen.
Once again, we call a recursive function, *addModelMatrix_Bridges*, that, by considering one by one the nodes of the tree, will return an array of transforms to apply to a unit quad in order to obtain the lines we need. The idea is to take a quad, rescale it to make it look more like a line, and finally translate it in the right spot. In this case, we directly return an array containing the model transform matrices for each bridge we want to build. This recursive function works as follows:

1. If this is the first time the function is called, the global vec3 variable *coordinatesNewNodeToDraw* will be set once again to (0, 0, -100). Don't be misled by the name: it still represents the coordinates we wish the center of a bridge to have.

2. We take the coordinates of the current node and save them in a variable, *posNode*.

3. If the current node has some children node, loop over them.
    (a) Take the coordinates of the current child node, and put them in a variable, called *childNode*.
    (b) Call recursively the function on this child node.
    (c) Once the recursive call has finished, we are sure that all the bridges have been generated correctly for the child node. This is not really important to build the current bridge, but it was worth stating to make explicit that, in the end, all bridges will be built.
    (d) If the child node is the very first one of the current node, build the model matrix that first scales the unit quad vertically so to fill the space between the two nodes, and then translate it into the midpoint between *posNode* and *childNode*.
    (e) Else, create a model matrix for a "half-bridge", that is, the same as before, but with half the vertical scale and in a position that is between the midpoint found before and *childNode*.

4. Once all the child nodes have been processed, we can create a horizontal bridge that will allow to connect all the child nodes from the 2-nd on to the vertical bridge created first. To do this, we save the coordinates of the last child node, *posLastChild*, and we build a model matrix that first scales horizontally the unit quad to fill all the space between the first and the last child node, and then translates it in the midpoint between *posNode* and *posLastChild*.

When performing a scale-up in one direction, a scale-down in the opposite also occours, so to effectively create a line segment. All the transform matrices created in this way are saved in the transforms array, that is then returned and can be used to correctly create the bridges we need.

After the nodes and the bridges, we should mention the text rendered on top of the nodes. However, at this point of the implementation, we simply used the *RenderText* function of the previous application, that *renders a quad for each letter, applying on top of it a texture.*

4.4 First input handling

If we draw a lot of stuff in a large scene, we also need a way to move around it. This is why we implemented a Camera class, found in the *camera.h* file. It is mostly taken and adapted from the learnOpenGL camera class. We constrained its movements along the x and y axis, allowing the user to also change the zoom with the mouse wheel. Changing the zoom effectively changes the projection matrix used. To move around the scene, use the WASD keys. To zoom in and out, use the mouse wheel.

4.5 Shaders v1

After all this setup, it was time to work on the shaders for the nodes. Nothing simpler: just define a vertex shader that, given the local coordinates of a vertex, and a model, view and projection matrices, multiplies all of them to obtain the Normalized Device Coordinates of each vertex. Each vertex has also a color that gets passed on to the fragment shader, which simply outputs it to the standard output vec4 *FragColor*. For the text, we used the same shaders as before.

4.6 Performance issues

With the implementation defined up until now, the first execution tests showed a massive slowdown and drop in performance, to the point that, in order to move around with the keyboard, more than a few seconds needed to be waited. This slowdown was of course proportional to the number of nodes given. Each node requires a quad to be drawn, and the number of bridges is proportional to the number of nodes, giving a minimum of $4n$ triangles to be rendered, if n is the number of nodes. On top of this, also the text needs to be accounted. Each node has *Exploration ID: x* drawn on top of it. "Exploration ID" is a 14 characters long string, which means that **each node** draws $\Omega(28)$ triangles. On top of this, in fact, we should add the number of triangles rendered for the effective Exploration ID number. We can clearly see that the number of primitives to render is starting to grow. We started to think of using some culling techniques, such as the View Frustum Culling, to reduce the number of primitives sent to the GPU. However, something was off: the application was slow even when just ten thousand triangles were rendered. Of course, they are *probably* not *few*, but **certainly** they are not **that many**! Otherwise many modern videogames would be unplayable. Moreover, we are not rendering a complex scene with different strange models inside of it, and neither the shaders to that much stuff: not even illumination models are used. This sense of "something's off" led us to make some researches online, because it really didn't make sense why *drawing so many simple primitives was causing such a big slowdown.* Eventually, we found the answer.
The problem does not lie on the fact that many primitives need to be rendered, but rather on the overhead given by the CPU-GPU communication. If thousands of quads are rendered, one by one, by calling *glDrawArrays* or *glDrawElements*, it means that we are telling thousands of times to the GPU, from the CPU, what buffer the data lies in, where the vertex attributes can be found, and so on. Therefore, the CPU to GPU bus becomes the bottleneck of our application. And since a priori the number of primitives drawn, though simple, can be indefinitely high, we have demonstrated that our application is not scalable, and will likely often froze because of this bottleneck.
The first solution that may come to mind is to artificially create ourselves a tree model: we put all the vertex data regarding the nodes and the bridges on a single VBO (maybe using an EBO too to avoid duplicate vertices), bind everything to a VAO, define the vertex attributes and we are done. Now we would only need one single *glDrawElements* call. The problem with this approach is that we would need to build the model ourselves, which is quite complicated. Moreover, it's not just about quads representing nodes and bridges, which in the end are simple rectangles: we also have to render text, that uses textures.
But the most frustrating part about all of this was that what we wanted to do was not that complex. With this we mean that we simply want to render the *same* primitive (a node, a bridge, or a text) more than once, by just changing *some* parameters, like the model matrix and the texture coordinates. **There's no need to pass vertex data and vertex attribute data to the GPU for each quad rendered: they are always the same informations!** This is why, after the successful research done before that let us understand the reason behind the low performances, we began a second round of researches. Eventually, we found the solution on learnOpenGL: Instancing.

4.7 Instancing

The concept of Instancing is thoroughly explained in the learnOpenGL website, so we'll just cover the key concepts and explain how they helped us solve our performance problems.

4.7.1 Key concepts

With instancing, we can draw the *same* mesh data many times on the GPU with just *one* render call. We simply have to replace *glDrawArrays* and *glDrawElements* with their instanced counterparts: *glDrawArraysInstanced* and *glDrawElementsInstanced*. They have the same arguments as before except for an extra input integer that we will call *instance count*. It represents the number of times the mesh data should be rendered, each of them being an ***instance***. Each instance of the mesh data can be distinguished from the others by its id, from 0 to ($instance\ count - 1$), given by the built-in vertex shader variable *gl_InstanceID*. However, drawing the exact same primitive more than one time seems kinda useless, if we can't fine-control them. Surprise, we actually can, by using the *instanced arrays*. We define an instanced array as a vertex attribute, but the difference is that they are updated per instance, instead of per vertex. This means that we can take a mesh, send it *once* to the GPU, render many instances of it, and have *some additional data associated to an instance* (a set of vertices) that we can use to make it unique. The way in which a vertex attribute becomes an instance array is by adding a call to *glVertexAttribDivisor(x, 1)*, where x is the index of the vertex attribute (now an instanced array). This is called after all the classic setup of a VBO and its attributes (creation, binding and filling of a VBO, setting and enabling of vertex attributes, that of couse comprehends the instance array). One last thing we should mention is that the instanced arrays passed can be of many data types, but not just the standard int, float and such. Even vec3, mat3, vec4 and mat4 instanced arrays can be used.

4.7.2 Nodes and bridges rendering revisited

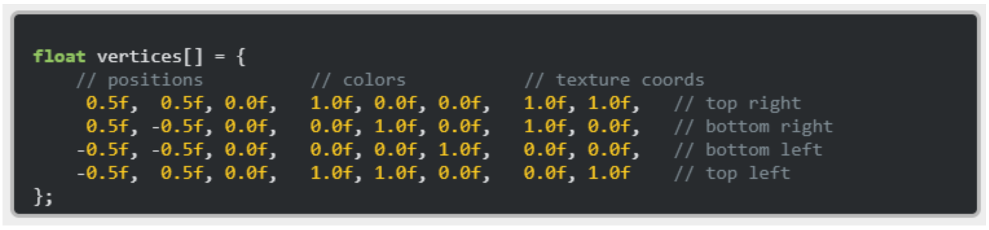
With instancing in our hands, we can finally render all the nodes, each of them being a quad, with just a render call. Now, the *addModelMatrix_Nodes* function will fill a mat4 array and a vec4 array, the former used to store the model matrices of each node, the latter their color. In the *getVAOWithDataToDrawNodesInTree* function we take these two arrays and convert them into instanced arrays that will be accessed by the *shader_node_in_tree* shader. In this way, each instance will have its own model matrix and its own color. The former will be used with the view and projection matrices (that instead will be the same for all the instances, which is why we declared them as uniforms) to determine the NDC of each node; the latter is passed on to the fragment shader. The same thing happens for the bridges, the only difference is in how the mat4 and vec4 arrays are built. This would solve all the performance problems... *if it wasn't for the text rendered on top of each node.*

4.7.3 The new text rendering

First of all, we realized that the *RenderText* function wouldn't have been of much use from now on, since each call to it results in a draw call. We need an instanced draw call for the text as well. Since there is a one-to-one correspondence between nodes drawn and the text on top of them, this shouldn't be too complex. Each string of characters will result in a certain number of quads drawn, all of them being on top of a certain node. Therefore, we can apply instancing to this as well, at least from the viewpoint of the model transform matrices.
To do that, we first of all need a function that takes as input a font and returns a bitmap font for it, so that we have the texture to apply to the quads. Since this is a bit complex to do, we leveraged on the code provided by Christian Behler, that allows to create a bitmap font, from a given font, using the FreeType library. We could have just used a random bitmap font found on google, however, this approach is more modular, giving full customization over the font used.
It is now time to initialize the VAO that will allow us to write all this text. This is done in the *getVAONodesText*. The idea is simple: say n is the number of nodes rendered. Given an array of n strings, we want each of them to be drawn on top of the corresponding node. To do that, we consider one string at a time, and create as many quads for it as the length of the string. Then, we create a fitting model transform matrix for that quad, and store all these matrices in a mat4 array. We will then convert this array into an instanced array that we can use in our shader. A problem, however, remains: *how do we actually put character textures on top of these quads, when we deal with instancing?*

4.7.4 The problem

Even though we could just skip to the solution, we think it's instructive to state the whole thought process we went through to get this right.
Up until now, if we wanted to put a texture on a model created by us (say, a triangle), we would have had to create a float array (the vertex data) containing, for each vertex, not only its local coordinates, but also its texture coordinates. What follows is an example image taken from the Textures chapter of learnOpenGL.



```
float vertices[] = {
    // positions          // colors           // texture coords
     0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f,   // top right
     0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 0.0f,   0.0f, 1.0f    // top left
};
```

After that, it's just a matter of defining the right way to interpret these vertex data through vertex attributes. Building the vertex data to render a quad with a specific character texture applied to it is fairly simple. The local coordinates of the four vertices can be the same of a node quad, since we can always rescale it however we prefer. While for the texture coordinates associated to each vertex, we developed a function called *getTextureCoordinatesOfCharacterInBitmap*. Given an input character, it used to return the correct (u, v) coordinates of the bottom-left and top-right vertices. For example, if the input was "*a*", the function would have returned:

1. 1/16 and 1/8 (bottom-left vertex);
2. 2/16 and 2/8 (top-right vertex).

The (u, v) coordinates of the remaining bottom-right and top-left vertices are easily obtained from those returned. These four (u, v) coordinates would then be put in the vertex data in the appropriate position, and later interpolated during rasterization to display the correct character texture of a given quad.
The problem in all of this? The texture coordinates are inherently a *vertex* attribute, and not an *instance* attribute. At this point, we thought we had reached a dead end. To solve this problem, we would somehow need to pass to each instance four pairs of coordinates, and then distinguish the specific vertex considered in the vertex shader. However, this is impossible: each vertex shader is isolated from the others. The *true* solution would be to somehow condense in a single variable, of whatever kind, the information regarding the texture coordinates associated to a vertex.

4.7.5 The solution

Well, we are proud to say, that we came up with a solution.
Reasoning about the texture coordinates that each quad needed, we noticed a pattern. Independently from the character to be drawn, there were certain properties that always held true:

1. The four texture points always described a region of area equal to $(1/16) * (1/8)$, which is the space allocated for every character in the bitmap;
2. Because of this, given the texture coordinates of the lower-left point of a character, say $(x0, y0)$, we can easily get the other three coordinates: $(x0 + 1/16, y0), (x0, y0 + 1/8), (x0 + 1/16, y0 + 1/8)$.

From these observations, we got an idea: let's initially set the texture coordinates to be the same for every quad that needs to render a character: $(0, 0)$ for the lower-left vertex, $(1/16, 0)$ for the lower-right vertex, $(0, 1/8)$ for the top-left vertex and $(1/16, 1/8)$ for the top-right vertex. Then, for each character, we compute its horizontal and vertical offsets, that are simply two values between 0 and 1 (actually, between 0 and $(1 - 1/16)$ for the x and between 0 and $(1 - 1/8)$ for the y) that must be summed to each initial coordinates to obtain the correct texture coordinates.
This procedure is actually pretty easy to visualize: imagine to have a "window" of width 1/16 and height 1/8 placed on the bottom-left part of the bitmap font. What is inside that window determines the texture to be applied to the current quad. With the offset values, we can freely move this window on the bitmap font, therefore having access to all the characters. And that's it!

To implement this, we modified the *getTextureCoordinatesOfCharacterInBitmap* into the *getTextureCoordinates**Offset**OfCharacterInBitmap*, that now gives a vec2 offset value instead of the four texture coordinates. We then build an instanced array out of these offset values, that is passed to the new shader *shader_text_in_space*. In the vertex shader, the (initial) texture coordinates are summed to their corresponding vec2 offset (*that is the same for every vertex of the same quad)*, and passed on to the fragment shader.

This concludes the Instancing part. Now, even tens of thousands of nodes can be easily rendered without affecting the performances, since the number of draw calls is now constant.

4.8 Second input handling

Now that the entire tree is efficiently rendered and we can freely move in the scene, the last piece of the puzzle is being able to click over a node to actually visualize it as we did in the Node Visualizer Application. To do that, we just need the world coordinates of the click (which we need to obtain) and the coordinates of the center of each node (which we have). Say c is a vector describing the center point of a node's quad, and (x, y) are the world coordinates of the click. Since we know that each node's quad is a unit quad, then the node was clicked iff $c.x - 0.5 < x < c.x + 0.5 \wedge c.y - 0.5 < y < c.y + 0.5$. This extremely simple intersection test can replace the standard Ray-Bounding Volume intersection test only because of the regularity of the models involved, as well as the two-dimensional environment we have built.
We now need the world coordinates of the click. To get them, we first register a mouse callback function (*mouse_button_callback*) to react to a left mouse click. From there, we can get the x and y coordinates of the pixel on window clicked using the function *glfwGetCursorPos*. Then, we can normalize these coordinates between -1 and 1 to get NDC. Since each change in coordinate system is just a matter of matrix-vector multiplication, if we want to go back to previous systems, we just need to use the inverse of the matrices applied. Therefore, we first of all multiplied the inverse of the projection transform with the NDC vector previously obtained. We then also apply the inverse of the view transform to the result to finally get the (x, y) world coordinates on which the click happened.
We now need to find the node on which the click happened, if there is any. The first idea would be to loop over all the nodes' centers, apply the intersection test, and return the node's Exploration ID of the node clicked (-1 if no node was clicked). The Exploration ID in fact is enough to run the Node Visualizer part of the project. However, it is obvious that this is an inefficient solution, and we didn't even bother implementing it. Therefore, we thought that we could rely on a Spatial Data Structure, such as a Bounding Volume Hierarchy, to speed-up this look-up. However, we actually already have a logarithmic-time complexity hierarchy to do this: the tree itself.
Each TreeNode contains, among the other informations, the position of the center of its quad and pointers to all its children, ordered from the leftmost to the rightmost. Therefore, we came up with the function *checkIfClickedOnNode* to find out if an intersection happened. Whenever a click happens, the function is called, passing as arguments the (x, y) world coordinates of the click and the TreeNode of the root node, that is the current node being checked:

1. If the current node is completely below y, return -1, as no node below the current one can intersect with (x, y).

2. Otherwise, check if the current node centered in c was clicked: $c.x - 0.5 < x < c.x + 0.5 \wedge c.y - 0.5 < y < c.y + 0.5$. If true, return the Exploration ID of the current node.

3. Otherwise, the search must continue downward. We don't however need to check all the child nodes, but only one of them. Since the child nodes are ordered from the leftmost to the rightmost, loop over them, and find the rightmost one that has its left edge to the left of x. If this node exists, call recursively on it. Otherwise, return -1.

The proposed algorithm has $\mathcal{O}(\log_k n)$ complexity, where n is the number of nodes in the tree and k is the average number of children a node has. Basically, the complexity is reduced to at most the number of levels in the tree.
When visualizing a node, pressing the **spacebar** key will make the application return to the tree visualization.

5 Future directions

The application works fine and scales well with the number of input nodes. However, we might decide in the future to change some things.

1. For now, the only text information shown on a node in the tree is its Exploration ID. If necessary, we might think of adding other informations to each node, such as the Primal Bound value. This however would just result in another *drawTextInTree* call, which is not a big deal.

2. Another "fancy" idea could be to change the color of the nodes depending on how much space was occupied in that solution: a white color for the empty root node, 100% green for the optimum one, and in-between for all the others, in order to show how much in depth the algorithm went on each sub-tree.

3. The last performance-related issue we have is in the initialization of the whole tree, namely, the computation of the model matrices of both the nodes as well the bridges. They are in fact computed at the beginning of an execution, during the Application Stage, before entering the render loop. It's usually not a big deal, since even with tens of thousands of nodes to process the application requires only a few seconds to compute the right matrices. However, in extreme cases in which we have millions of nodes, even some minutes might be necessary to setup everything. A possible solution could be a multithreading application. The idea would be to start a thread at the beginning and have it process each node, putting the nodes ready to be rendered on a vector of TreeNodes. Then, the rendering loop would only render the nodes present in such vector. Over time, more and more nodes would be processed and made available to be rendered, updating in real time the quads rendered. It's true that in this way not the whole tree is ready once the application is launched, but that's not really important. In a real-case scenario, not only the user would have in a bunch of seconds tens of thousands of nodes ready to be seen, but also, since the nodes are processed in Exploration ID order, the nodes not immediately available would be the ones later explored by the B&B algorithm, that would probably be viewed anyway in a second moment.