5.5 Algorithm for Finding Cells

To let the algorithm find a cell the user has to select a "halfface". Because in OVMVR halffaces are not a single compoonent that is shown, but only the face on its own is present, the user is placing the PlacementChaperone on the preferred side of a face, so the direction to which the cell should be created is known to the algorithm, simulating the behaviour if a certain halfface would have been selected. Therefore when there isn't a cell on the preferred side of the face bot on the other one, the algorithm will terminate by finding the other cell. But because it would have fully searched for the one that was preferred this behaviour can be disregarded, because the user still has the control over whether the cell should be created or not. The algorithm will either terminate by highlighting a found cell containing the selected face in red or won't mark anything at all and indicates that it has not found anything.
To visualize the following steps, the following mesh is considered (most of the highlighting is not shown in the editor at runtime; only the result is marked in red):

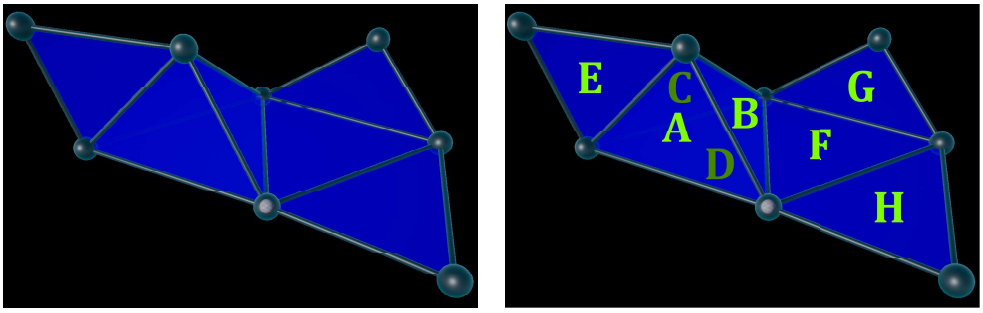


Figure 5.1: Initial Mesh

The user wants to create a cell by selecting face **A**, which is then passed as a parameter to the *CalculateCell function*:

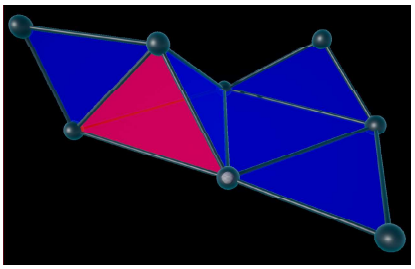


Figure 5.2: Selected Face

First this selected face is added to the *CheckedArray* to keep track of the already processed faces to save time and computational power. Then this face is checked whether it is "fully enclosed" in the following manner:

```
for each edge boundary of currentFace :
        if edge is boundary of at least two faces :
                add currentFace to FaceArray
                check faces adjacent to currentFace ...
        otherwise :
                return false
```

Because in this example the selected face is "fully enclosed" it is added to the *FaceArray* - which in the end will hold all the faces of the valid cell (none if no cell was found). Then all the adjacent faces of the selected one are checked recursively one after another whether they will lead to a valid cell or not. The adjacent faces are the following:

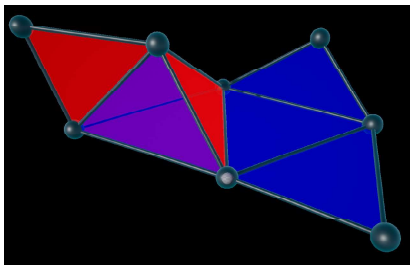
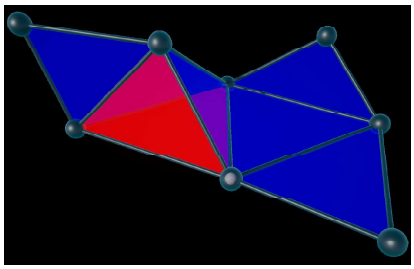


Figure 5.3: Adjacent faces of the selected face

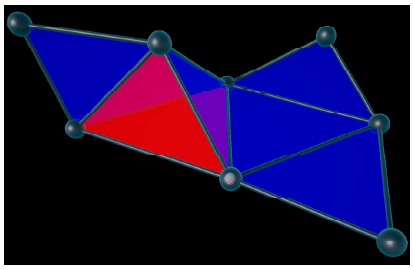
These faces then are sorted by prority:

1. face which is adjacent to the most faces which are already contained within the *FaceArray* (synonym with the future cell)
2. sharpest angle between the face and the previous checked face (also considering which side of the initially selected face was preferred)

Therefore the first face that is checked is one that is contained within the terahedron, because the angles to the initial face are sharper than the angle to the face to the left (**E**); for this example we will say that the bottom face of the tetrahedron (**D**) is the first one in the array after sorting. This face is then passed as a parameter to the *CalculateCell function* and is added again to both *FaceArray* and *CheckedArray*, because it is also "fully enclosed".



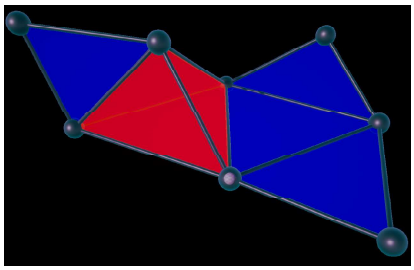
(a) Content of FaceArray



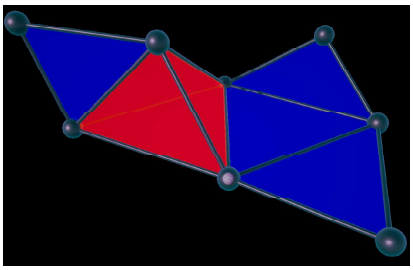
(b) Content of CheckedArray

Figure 5.4: Contents of both arrays

Again the adjacent faces are fetched and sorted. Because the front and bottom face are contained in the *FaceArray* the left and right side of the tetrahedron (**B** and **C**) have a higher priority to be checked. After one was checked and added to both arrays, the last face of the tetrahedron has a higher prority than all the other faces therefore after three steps the *FaceArray* and *CheckedArray* are containing the following faces:



(a) Content of FaceArray

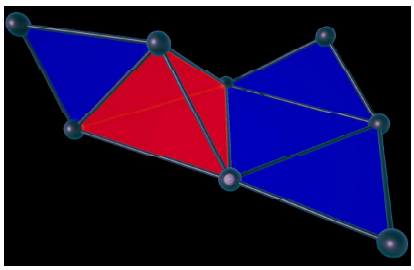


(b) Content of CheckedArray

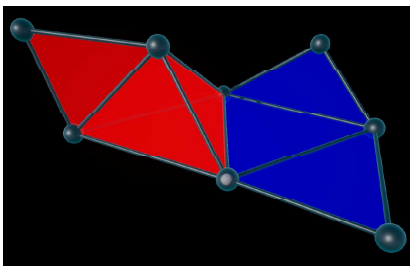
Figure 5.5: Contents of both arrays

The recursion algorithm is not over so if the last face that was checked was face **C** its adjacent faces will need to be checked as well. All the faces of the tetrahedron have the same priority but because they are already part of the *FaceArray* they are skipped. The face on the far left (face **E**) is therefore the only candidate available where the cell could continue further, but because it is not "fully enclosed" it is removed from the *FaceArray* and it will count as checked. Because all of the edges of face **C** have one additional face that can lead to a valid cell the face will not be removed from the *FaceArray* and the algorithm will return true for this particular face:

```
for each edge of face X:
        if all adjacent faces return false :
                return false
return true
```



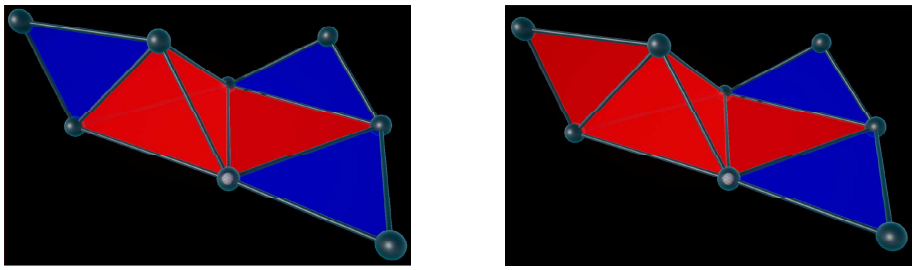
(a) Content of FaceArray



(b) Content of CheckedArray

Figure 5.6: Contents of both arrays

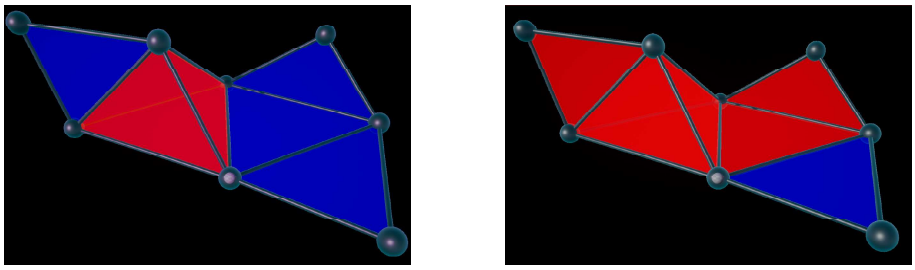
Next the other checks for the right side of the tetrahedron are made. Again the bottom and front part of the tetrahedron are skipped because they are still in the *FaceArray*, and the algorithm returned true for the left side of the tetrahedron. The last thing to check is whether the face adjacent to the right (face **F**) could be a continuation of the cell. Because it is "fully enclosed" it will be added to the *FaceArray*:



(a) Content of FaceArray **(b)** Content of CheckedArray

Figure 5.7: Contents of both arrays

Its adjacent faces are then checked. The faces that are part of the tetrahedron are again skipped. Because faces **G** and **H** are not "fully enclosed" their checks will return false. Because the face does not fulfill the criteria that each of its edges have exactly 2 faces that fulfill the "fully enclosed" criteria it is removed from the *FaceArray*. Because one side already returns with a false the other side does not need to be checked. Therefore the check for the enclosed face on the right will return false, but because the right side of the tetrahedron (face **B**) fulfills the criteria that there are exactly two "fully enclosed" faces on each of its edges, it will not be removed and the algorithm returns true for it. The same is happening for the bottom and the front face of the tetrahedron. The algorithm terminates with the four faces of the tetrahedron selected as a valid target to create a cell.



(a) Content of FaceArray **(b)** Content of CheckedArray

Figure 5.8: Contents of both arrays after the termination of the algorithm