Manipulating the cameras viewing parameters

*Static cameras in the universe*

The default function for setting the cameras position in the world has already been explained in Chapter 1. The function

```c
VRT_SetDefaultCamera(float ex, float ey, float ez, float cx, 
                      float cy, float cz, float ux, float uy, 
                      float uz);
```

sets explicitly the position of the observer \(E = (ex, ey, ez)\) in the world and the center of the focus \(C = (cx, cy, cz)\). Therefore the viewing direction is defined by a vector pointing from \(E\) to \(C\). In addition, the roll of the camera around the viewing axis is defined by the up-vector \(U = (ux, uy, uz)\). For mathematical correctness, the up-vector \(u\) should always be perpendicular to the viewing direction vector. It is however, not always easy for the application programmer to figure out, how the correct up-vector should be for a given viewing position \(E\) and target viewing point \(C\). Therefore, in practice, the specified up vector indicates approximately towards the actual up direction.
Setting the camera with above-mentioned function also requires that positions and directions are known with regard to the global universe coordinate system.

For those reasons, this function should be used, when the camera viewpoint in a scene is static i.e. it does not vary so much over time, or is restricted to a certain number of well known pre-defined parameter configurations.

**Animated cameras**

The static view on a 3D scene is often typical for applications, where the user is studying intensively an object from a given perspective. Careful analysis of 3D scenarios often does not lend to simultaneous camera manipulation.

However, in so called immersive applications, the users tend to virtually move themselves within a 3D scene and therefore do intensively control and change their viewing position.

In order to ease handling of moving cameras, VRT allows the application programmer to attach the scene default camera to any node in the scene graph hierarchy. Whenever VRT encounters, that the default camera is being attached to a node it will use the nodes current attitude with regard to the universe coordinate system, in order to automatically calculate and set the camera parameters. Attaching a camera to a node is being done using the following function:

```c
VRT_AttachCameraToNode(VRT_Node *node);
```

Hereby `node` must point to a valid node in the scene graph hierarchy or must be `NULL`. Whenever `NULL` is passed to this function as parameter, automatic camera tracking is disabled, and the previously set default camera values are being used to configure the camera. Also, whenever a call to `VRT_SetDefaultCamera(...)` is issued, any previous camera binding to a node is being discontinued!

The node passed to the function `VRT_AttachCameraToNode(...)` can be a node which actually represents a geometry in the scene. The nodes attributes or behavior is not affected by camera binding. It is however more common, to bind cameras to a carefully chosen construction of a node hierarchy, which only fulfills the purpose of modeling the camera animation.

Attaching a camera to a node will cause the camera to automatically track the node in the following way:

- The viewing position is bound to the node’s origin
- The viewing direction is downwards the negative z-axis of the node
- The up-vector is up the positive y-axis of the node

Not always, the local attitude of a node which shall be tracked by the camera is suitable for controlling the cameras parameters. In those cases, additional nodes must be added to the node of interest, which only fulfil the purpose of re-arranging the cameras permanent attitude with respect to the node of interest. Here are some examples:
**Following with a moving object**

Suppose the camera shall be positioned on the roof of a car, which is moving through the scene. Suppose also that the geometry of the car which is set to the car’s node faces towards the negative x-axis of the car’s node.

Attaching the camera immediately to the car’s node, will therefore have the effect of having the camera viewing through the right side window of the car, no matter how the car moves. This is, because the camera faces downwards the negative z-axis of the car’s node, which is to the right of the negative x-axis, where the front side of the car is facing to.

The solution to the problem is to define a node (lets call it `cnode`) which is a child of the cars node. When the camera is attached to `cnode`, and `cnode` is rotated about 90 degrees around the y-axis, the camera will look to the correct direction.

Note also, that this helper node can be manipulated while the car is driving through the scene. The camera can therefore travel with the car, and pan to the left and right, depending on were the “virtual driver” driver is looking toward.
**Following behind a moving object**

Consider the same car scenario as above, this time however, the camera is supposed to follow the car from behind all the time at distance $d$. Again, this can be solved by using an additional helper node. This time, however, $cnode$ is translated along the positive y-axes in relation to the cars node about the amount of $d$ so that the cameras position is $d$ behind the car (remember: the car is facing forward towards the negative x axis) of its node. In addition to moving $cnode$ behind the car, also the direction of $cnode$ must be adjusted such that the negative z-axis of $cnode$ points towards the direction of the cars negative x-axis. As above, this can be achieved by rotating $cnode$ about 90 degrees around the y-axis.

**Spotting around a moving object**

The camera animations above are rather simple and have been possible to solve with only one helper node. In practice, two or even more nodes can be required to model the relation between a camera and an animated object in a realistic way. To demonstrate this, consider that the camera in the above mentioned examples is now supposed to fly around the car at a certain height above the car, while the car is travelling through the scene. The cameras circular course above and around the animated car can be modeled very easily using two nodes, which are linked to the car’s node. Let’s suppose $cbase$ is a helper node attached to the car’s node, and $cnode$ is also a helper node attached to $cbase$. In order to accomplish a camera +position above and distant from the car, we can move $cnode$ along the z-axis and up the y-axis about equal amounts of $d$. The statement would look like this:

```plaintext
VRT_NodeTranslate(cnode, 0, d, d);
```

In this position, however, the camera is looking across the object to be spotted, which is still being at $(0,-d,-d)$ relative to $cnode$. Adjusting $cnode$ negative z-axis only requires a rotation of $cnode$ about $-45$ degrees around the x-axis of $cnode$.

```plaintext
VRT_NodeRotate(cnode, -45, 0, 0);
```

With those two commands, $cnode$ has been brought into a rigid relation to $cbase$ and also the car’s node, such that the camera (being attached to $cnode$) is located aside and above of the car’s center and is facing downwards upon the cars center.
In this situation, the role of $cbase$ becomes evident. In order to have the camera rotating around the car, $cbase$ now can be rotated continuously around its y axis, while the scene is animated. The rotation of $cbase$ does not affect the cameras upward position and downward facing attitude but does change the rotational angle of this configuration in relation to the car’s node.
Freezing the camera in the current attitude of an animated object

Suppose that the application requires the camera to travel with a specified node anode until, on user intervention, the camera shall be frozen in the nodes current attitude, while animation continues to proceed.

While the camera is travelling, naturally the camera would be bound to anode with the technique described above using the VRT_AttachCameraToNode(...) function.

As soon as the camera shall be frozen in the current attitude of anode, the camera must be detached from anode and either be

a) attached to another node, with anodes current attitude, or
b) set explicitly to appropriate default viewing parameters.

Version a) can be appropriate, whenever after a while the camera is supposed to be further animated independently from anode. In this situation it is worth to create a new node nnode to which the camera will be attached, and which later on will be transformed. The tricky issue is now to align nnode with anode at the time when the camera shall be frozen. Without making any assumptions on anodes level in the scene graph hierarchy, it can generally solved in the following way:

Retrieve anodes attitude matrix with regard to the universe, which represents all concatenated matrixes along the node-path from root to anode.

Create a new node nnode, which is linked to root. And set nnodes local transformation matrix to anodes global attitude matrix. Since nnode is linked immediately to root, its local attitude matrix yields effectively the same orientation as anodes, no matter how many nodes are linked together from root to anode. The code fragment to accomplish this would read like follows:

```c
void freeze_camera_a()
{
    float antm[16]; // transformation matrix of anode

    VRT_NodeGetTransform(anode,antm);
    If (nnode == NULL) VRT_NodeNew(root,"new camera node");
    VRT_NodeSetLocalTransform(nnode,antm);
    VRT_AttachCameraToNode(nnode);
    .
    .
}
```

Version b) can generally be recommended, as long as no further camera animation starting from the frozen attitude is intended. Also, if the camera is supposed to follow the other nodes later on with the option for iterated freezing procedures.
Since explicit camera parameters are set with regard to the global reference frame of the universe, it is required to transform current camera position, viewing direction and up-vector to global parameters before calling the function `VRT_SetDefaultCamera(...)`.

Note that the camera attached to an animated node `anode` faces downwards the negative direction of the z-axis of `anode` while being positioned at the origin of `anode` with regard to the universe. The up-direction of the camera is defined by `anode`’s y-axis. Note that `VRT_SetDefaultCamera(...)` requires an absolute camera position, an absolute target point and a vector in universe coordinate system. Those can be calculated by transforming arbitrary points from `anode`’s local coordinate system into the global universe coordinate system. An arbitrary target point `tp` on the line of sight of the tracked camera is e.g. `{0, 0, -1}` in `anode` coordinate system. The cameras position `cp` in `anode` coordinate system is `[0, 0, 0]`. An arbitrary point `up` on the camera’s up-vector would be for example `[0, 1, 0]`. Transforming `cp` and `tp` into global universe coordinates, requires their transformation with `anode`’s attitude matrix. Calculating the up-direction vector `uv` of the camera in world coordinate system, requires the transformation of `up` into world coordinates and then calculating the direction from `cp` to `up` i.e. `uv = up - cp`. Putting this all together would look like this:

```c
void freeze_camera_b()
{
    float antm[16]; // transformation matrix of anode
    float cp[4] = { 0, 0, 0, 1};
    float tp[4] = { 0, 0,-1, 1};
    float up[4] = { 0, 1, 0, 1};
    float uv[4];

    VRT_NodeGetTransform(anode,antm);
    VRT_TransformVertex3D(cp);
    VRT_TransformVertex3D(tp);
    VRT_TransformVertex3D(up);

    uv[0] = up[0]-cp[0];
    uv[1] = up[1]-cp[1];

    VRT_SetDefaultCamera(cp[0], cp[1], cp[2],
                        tp[0], tp[1], tp[2],
                        uv[0], uv[1], uv[2]);

    // NOTE: VRT_SetDefaultCamera automatically resolves
    //       camera binding to a currently tracked node !
    
}
```
Note, that the function sketched here must be called **once** at the time when the camera shall be frozen in the current position of an animated node, to which the camera is attached.

**Tracking the view from object A to B**
Having understood the mechanisms from `freeze_camera_b(...)`, it should now be easy to realize continuous binding of the cameras position to node `anode` and the cameras viewing direction from `anode` towards another node `bnode`. For this purpose, a function `update_camera_a_to_b(...)` can be defined, which must be called every time the simulation callback function is entered and while this camera tracking is desired. For simplicity reasons, let’s assume that we are facing from `anode`’s origin towards `bnode`’s origin.

```c
void update_camera_a_to_b(VRT_Node *anode, VRT_Node *bnode)
{
    float tm[16];
    float a[4] = { 0,0,0,1};
    float b[4] = { 0,0,0,1};
    float uv[4] = { 0,1,0,1};

    VRT_NodeGetTransform(anode,tm);
    VRT_TransformVertex3D(a);

    VRT_NodeGetTransform(bnode,tm);
    VRT_TransformVertex3D(b);

    VRT_SetDefaultCamera(a[0], a[1], a[2],
                        b[0], b[1], b[2],
                        uv[0], uv[1], uv[2]);
}
```

Note: This function must be repeatedly called from the simulation callback function whenever camera tracking is desired. Also, we suppose a constant camera up-vector of [0,1,0], which for most simulation settings should be the default upward direction. However, when the current viewing direction determined by `anode`’s and `bnode`’s position in the world approach `uv`’s direction, camera roll will be undefined!
Advanced object collision detection techniques

Checking for arbitrary object collision
Collision detection of objects is not a feature which is enabled in VRT simulations all the time. Since it a very computing intensive task to check all geometries in a scene for collisions, it is worth to reduce the number of collision tests to those objects which are actually of interest. Usually, only a few objects in a scene are considered for collision detection testing. Therefore, VRT leaves it up to the application programmer to actively investigate collisions on demand. The easiest way for the programmer to detect object-to-object collision is to use the VRT function

\[ \text{VRT\_NodeIntersectNode(VRT\_Node *sn, VRT\_Node *tn)}; \]

The function basically executes a bounding sphere intersection test for the geometric objects attached to the respective nodes. The function automatically takes care for the objects’ attitude with regard to the world coordinate system i.e. for all transformations along the nodepaths to the nodes. Observe that the bounding sphere intersection test method is quite efficient in terms of minimized calculation efforts. The test requires also, that the bounding sphere parameters of the geometric objects linked to the nodes be known. The latter is usually the case, whenever a geometry has been locked after generic build-up. The bounding sphere intersection test results are quite unspecific in cases of asymmetric or elongated objects i.e. positive test results do not necessarily mean actual intersection. In order to specifically find out, if and where the surfaces of two objects actually intersect, detailed polygon intersection testing can be performed on all possible polygon pairings of the two objects.

Detailed per-polygon intersection testing
The brute force method of testing polygon pairs requires two nested loops where every polygon of object 1 is tested for intersection with each of object 2s polygons. The elementary VRT function to test for polygon intersection is:

\[ \text{VRT\_IntersectTriangleTriangle(float *v1, float *v2, float *v3, float *u1, float *u2, float *u3)}; \]
The function performs an explicit test on two triangles, which are specified by their three vertices. The vertex parameters \(v_1, v_2, v_3, u_1, u_2,\) and \(u_3\) are pointers to three cartesian coordinates.

The function does not make any assumption on the objects attitude. In other words, the vertices of each triangle of a polygon of two objects must first be transformed into a common coordinate system.

Note also, that all polygons of an object other than triangles must be broken down into sets of triangles before. Then, the pseudo-code for a brute force collision detection method would look like this:

1. get attitude matrix \(m_1\) of object (node) 1
2. get attitude matrix \(m_2\) of object (node) 2
3. for any polygon \(p_1\) in object 1
4.   for any polygon \(p_2\) in object 2
5.       transform vertices \(v_1...v_3\) of polygon \(p_1\)
       into world coordinates using \(m_1\)
6.       transform vertices \(u_1...u_3\) of polygon \(p_2\)
       into world coordinates using \(m_2\)
7.       TriangleTriangleIntersection\((v_1,v_2,v_3,u_1,u_2,u_3)\)

Note that the attitude matrix of the objects (nodes) as retrieved using the function \texttt{VRT\_NodeGet\_Transform(...)} transforms vertices from the nodes local coordinates into -coordinates.

As can be seen easily, it is quite inefficient to transform both the vertices of object 1 and object 2 into the world coordinate system. Instead, one could transform the vertices from object 1 immediately into the coordinate system of object 2. Then, the transformation of vertices in object 2 is obsolete. Transforming from node 1 to node 2 requires first a transformation from local coordinates of node 1 into world coordinates using \(m_1\). Then a transformation of global coordinates into local coordinates of node 2 is performed. For the latter transform, the attitude matrix \(m_2\) of node 2 matrix must be inverted. The pseudo-code for this modified version would read like follows:

1. get attitude matrix \(m_1\) of object (node) 1
2. get attitude matrix \(m_2\) of object (node) 2
3. invert \(m_2\)
4. concatenate \(m_1\) with \(m_2\)
5. for any polygon \(p_1\) in object 1
6.   for any polygon \(p_2\) in object 2
7.       transform vertices \(v_1...v_3\) of polygon \(p_1\)
       into coordinate system of object 2 using \(m_1\)
8.       TriangleTriangleIntersection\((v_1,v_2,v_3,u_1,u_2,u_3)\)
Refined object collision using adapted bounding shapes

Although being the most exact test for collision testing, the polygon pair intersection test is a very time consuming procedure and is, depending on the number of objects and the number of triangles to be tested, not always recommendable for interactive graphical simulations.

An acceptable solution to this dilemma provides the following two methods:

**Level-Of-Detail (LoD) management:**

This approach aims at representing objects with differently well resolved geometries. Depending on the task of the application, the highly resolved geometry is for example used for graphical output in the rendering stage, whereas a lower resolved version of the same object (with a significantly lower polygon count) is used for collision testing.

The lower resolved version is not intended to yield visual correctness, instead it must be chosen depending on the specificity and/or speed of the collision test intended.

Since LoD management highly depends on knowledge about the scene, VRT does not automatically support LoD management. Instead it is the task of the application programmer to handle multiple geometries for rendering and collision testing, depending on the task to be solved. However, VRT does supply a function to diminish the polygon count of an object, while trying to preserve the objects specific shape. The utility function

```c
int VRT_GeometryDecimate(VRT_Geometry *geom,
     float fuzz_factor);
```

tries to simplify the geometry by combining adjacent polygons in neighborhoods with little surface curvature. The `fuzz_factor` is a normalized value in the range between 0.0 and 1.0 which tells the threshold for the curvature measure. Typical values for `fuzz_factor` to start with are 0.95 or 0.9. The function returns the number of polygons which have been eliminated in the decimation process. It can be called repeatedly for a geometry until no polygons could be eliminated.

The effect of LoD management for collision detection is to have an implied geometric bounding volume which suffices object shape as much as desired.

Note that, apart from the automatic mesh decimation provided by VRT, much more efficient bounding geometries can be constructed by the object designer in the modeling stage.

**Approximation with clouds of spheres:**

The above discussed described method of using different levels of detail for collision detection falls back upon the generic triangle-triangle intersection test, which still is relatively expensive in terms of CPU cycles.

An alternative approach to this is, to have a collection of different bounding spheres approximate an object. Sphere-to-sphere intersection testing is significantly faster than triangle-triangle testing and therefore it
can be worth to construct a sub-tree with spheres in the scene-graph which approximates the object to be tested. Note, that all of the nodes in this sub-tree then must be checked for collision (using \texttt{VRT\_NodeIntersectNode(\ldots))}.

\textbf{Integrating domain specific knowledge into collision testing}  
The best results in term of runtime efficiency can be accomplished if a-priori knowledge for domain specific situations is combined with the elementary collision tests as described earlier (node-to-node, ray-node, triangle-triangle, ray-triangle). In addition, this a-priory knowledge on the simulation scenario should be used in the process of scene modeling in order to benefit from a number of techniques described here.

\textbf{Hotspots and selective areas:}  
In quite many applications, it is often only of interest to know whether a specific area of an object has been hit or not. Suppose for example a simulation of the drilling procedure in dental tooth preparation. For the visual appearance of the simulation it is essential, that the tooth surface be modeled as detailed as possible, while at the same time also the drill with the hand piece must appear highly realistic. However, when it comes to collision detection under the actual simulation, it is sufficient to know when the drills tip only hits the enamel of the tooth. Therefore, a hot-spot object like a small sphere can be linked to the drill object and be embedded into the drill’s tip. Collision detection needs then only to take care about the bounding sphere of the hot-spot object against tooth enamel.

Note that this sort of selective sub-area of an object can be implemented either with sub-nodes containing suitably shaped and arranged sub-objects (lending to node-node testing). Or it could be implemented by identifying the few polygons in the drill geometry which form the drills tip (lending to polygon-polygon tests).

Hot-spots and selective areas should be used whenever possible and are often suitable when user-to-object interaction is supposed to be implemented.

\textbf{Object collision in global co-ordinates:}  
Many simulations make an assumption of a ground level, which is the reference for simulations (the green of a soccer field, sea water level…). Objects are often not allowed to penetrate beyond the other side of those ground levels. As long as the ground of a simulation is co-incident with for example the xz-plane of the global coordinate system, penetration/hit test is particular trivial and fast.

For example the lowest vertex of an object O to be tested will be transformed into global co-ordinates by transforming this vertex with the nodes current attitude matrix (retrieve using \texttt{VRT\_NodeGetTransform(\ldots)}). The y-coordinate of transformed vertex is then compared to the elevation level desired (zero in this case). This test is considerably faster than comparing each polygon of O with each polygon of the ground geometry.
Similar situation can be found for object collisions with other plane geometries, which preferably should be aligned with base planes in the global context. For example in the simulation of a squash game, all the collisions of a ball with the walls and the ground can be efficiently implemented using the level testing of global coordinates. Note, however, that geometric modeling is chosen adequately such that the walls of the squash court coincide with base planes and planes co-planar with them.

**Ground hit test using height fields:**
The situation becomes more sophisticated, if the ground level has some structured elevation. In the case of a golf simulation, it is unlikely that the golf course would be modeled sufficiently well with a planar polygon. In this case it is advisable, to create a landscape representation, which is based upon elevation maps or height fields. A height field is a 2D rectangular array (matrix) of elevation values. The indexes to this two-dimensional matrix can be aligned with the x-, and z-coordinate in world coordinates. The elevation value of the ground at a given xz-position in world coordinates is then given by the corresponding entry in the matrix. In order to visualize these elevation maps, appropriately connected polygon-meshes must be created, which have the corresponding height values from the elevation map. VRT supports creation of those ground geometries from height maps by providing the function

```c
VRT_Geometry *VRT_GeometryFromPatch(VRT_Patch p);
```

This function returns a geometric representation of the matrix of elevation values supplied with the patch `p`. See the reference manual for a more elaborated description of this function. The only task of the application programmer is, to convert global coordinates (usually in floating-point format) of a point to be tested into appropriate matrix indexes (integer-value and range checked matrix index). Then the stored elevation value in the matrix can be looked-up and compared with the points actual y-value in world coordinates.

**Note:** Scaling of elevation map geometries must be taken care for both when transforming world-coordinates into matrix indexes, as well as when comparing world-coordinates with elevation values.

**Semantic maps**
An extension of height fields can be made in order to spatially encode events of different semantic level. The technique is the same as in height maps, but it is not spatial elevation maps which are stored in the matrices. Instead it could be semantic attributes coded into a map. Imagine a maze in a game inside which a player is supposed to navigate. Both the property of being a wall or being corridor can be encoded in a matrix which is aligned with the xz-plane. Depending on the players’
current xz-coordinate, the corresponding matrix indexes are calculated, and looked up to find out if the player hits the wall. Many more events could be encoded in that way. Bonus points spread out in the maze, trapdoors and sensitive teleporting zones are only a few examples of what could be identified by looking up the users’ position within the semantic map.

Note: Semantic maps must be resolved sufficiently high in order to allow for fine enough discrimination of states/events.
Advanced picking

Object picking
As has been explained in chapter 4.4, the function

\[
\text{VRT\_IntersectRayNode}(\text{float } *p, \text{ float } *d, \text{ VRT\_Node } *\text{anode, float } *t, \text{ VRT\_HPlg } *\text{hit\_poly});
\]

can be used to identify whether a ray (given in parametric form) intersects with a node. It shall be added that the intersection test holds true for any ray definition in the scene including pointing rays from the observer. The Ray-Node intersection test is performed in two steps. First, a bounding sphere test is performed and, if this one is positive, all polygons in the object attached to the node are tested for ray-polygon intersection.

Since there might be many polygons situated on the direction of the ray, there is an ambiguity problem in case of a positive test result. VRT, however, sorts out all but one polygon, and returns only the hit polygon which is positioned closest to the origin of the specified ray.

For user initiated picking of objects, a picking ray must be calculated emanating from the observers eye, and passing through the mouse pointer position on-screen and into the scene. The VRT function to be used to calculate picking rays, as already described in chapter 4.4, is

\[
\text{VRT\_CalcPickingRay}(\text{int } x, \text{ int } y, \text{ float } *\text{rayPos, float } *\text{rayDir});
\]

Selective object picking
As with object-object collision testing, it can be advisable to perform ray-polygon intersection testing on own initiative rather than using the high-level function \text{VRT\_IntersectRayNode}(\ldots). As is the case for selective object-object collision, hot-spots should always be used whenever possible also in user initiated picking. For this purpose, the user must store handles to sensitive polygons of objects which can be tested explicitly for ray intersection using the following two VRT functions:

\[
\text{VRT\_IntersectRayTriangle}(\text{float } *p, \text{ float } *d, \text{ float } *v1, \text{ float } *v2, \text{ float } *v3, \text{ float } *t);
\]

\[
\text{VRT\_IntersectRayQuad}(\text{float } *p, \text{ float } *d, \text{ float } *v1, \text{ float } *v2, \text{ float } *v3, \text{ float } *v4, \text{ float } *t);
\]

As holds true for the triangle-triangle intersection test, also here the vertices of triangles or quadrilateral polygons are passed explicitly with
regard to the global world reference system, as is the case for the ray position and direction. Therefore, polygons of objects linked to a node in the scene graph hierarchy must first be transformed into the global coordinate system, before ray-polygon intersection can be performed. If the test is positive, the parameter $t$ returned by the functions indicates the distance from $p$ to the hit-point along the ray $d$. The factual hit point in regard to the global world coordinate system can again be calculated as described in chapter 4.4:

$$\text{hitpoint} = p + t*d$$

// note: $p$ is a 3D coordinate, $d$ is a 3D direction vector!

**Texel picking**

Picking objects on the polygon level might be sufficient for most application areas. However, there are some specific situations, where selection tasks must be performed even on a per-fragment level or in other words on an on-screen pixel resolution. Here are some examples where this can be required:

**Stenciled textures**: Whenever textured polygons with stenciling are used to shape or model filigrane objects, it is not sufficient to be able to tell whether a user mouse click has hit the polygon. In the case of a tree masked texture, only a relatively small proportion of the polygon upon which the texture has been mapped represents visible structures in the scene. A positive hit-test against the polygon does not mean that the wood of a tree actually has been hit. Instead, the users pointing ray could actually point into a masked area, which is supposed to represent air (using appropriate stenciling techniques as described in chapter 3.4). Correct collision testing requires here to identify the texture coordinates of the hit position within the texture mapped polygon. Using the texture co-ordinates, the actual alpha channel and/or color component of the texture can be analyzed in order to identify what kind of structure was hit.

**Selectable textures**: Textures can also be used in a 3D environment to exhibit texts and selective areas on objects. For instance, a multiple choice question to be answered by a user in a 3D environment could be put as a texture upon a polygon in the scene. The clickable answering buttons could be color-coded with well known and distinguishable color-codes. A user click upon the questionnaire texture will then lead to the identification of the hit texel color. Based upon the selected color (i.e. hit field) corresponding actions can be taken.
Figure 5-1: Question answering dialogue in a 3D environment. The dialogue is implemented using a texture mapped polygon. The answering options are color coded with slightly different RGBA quadruples, which can be evaluated after the texture coordinate of a hit picked texel has been identified.

Drawing areas in 3D scenes:
Another scenario would be a white-board within a 3D environment upon which the user can draw arbitrarily. It could be implemented with texture mapping. Any user hit of the white-board polygon will result in setting the corresponding texture element with an appropriate drawing color and redrawing the texture which is mapped upon the polygon.

The elementary VRT function, which supports this type of detailed texel-selection, is:

```c
int VRT_IntersectRayTriangleUVT(float *p, float *d, float *v1, float *v2, float *v3, float *uvt);
```

Basically this function does not immediately return texture coordinates of the hit. Instead, it returns the solution parameters for the ray-polygon intersection test.
In the case of a ray-polygon hit, the hit-point’s cartesian coordinates can be reached by either traveling from \( p \) towards the direction \( d \) about the amount of \( t \), where \( t \) is one of the solution parameters returned by the function. Alternatively, the hit point \( \text{hit} \) can be reached by starting from vertex \( v_1 \) following the direction of vector \( s_1 \) about the amount of \( u \) and then following the vector \( s_2 \) about the amount of \( v \). Where \( u \) and \( v \) are the other solution parameters returned by the function.

What holds true for the cartesian coordinate space holds also true for the texture coordinate space. Therefore, the same approach can be applied to the polygons actual texture coordinates as illustrated below:

The same parametric solution as retrieved from the ray-polygon intersection test can be applied to texture coordinates and texture-coordinate vectors. Since there is no texture data available for the ray, the texture’s hit point is solely based upon the texture coordinates defined in the vertices of the polygon. Accordingly, the texture’s hit point is calculated by starting from \( t_1 \) following towards the texture-direction.
vector \( \mathbf{dt}_1 \) about the amount of \( u \) and then following the texture-direction vector \( \mathbf{t}_2 \) about the amount of \( v \).

Note that \( \mathbf{t}_1 \), and \( \mathbf{t}_2 \), and \( \mathbf{t}_3 \) are the texture coordinates in the polygon’s vertices whereas \( \mathbf{dt}_1 \) and \( \mathbf{dt}_2 \) are differential texture vectors pointing from \( \mathbf{t}_1 \) to \( \mathbf{t}_2 \) or \( \mathbf{t}_1 \) to \( \mathbf{t}_3 \) respectively. The source code fragment below puts this abstract derivation into concrete program statements. For simplicity reason lets suppose, that \( \mathbf{gtriangle} \) is a geometry, which contains only three vertices, which make up one triangle. Also let \( \mathbf{ntriangle} \) be the node in the scene graph, to which \( \mathbf{gtriangle} \) is linked:

```c
void TexelHit(int x, int y)
{
    float rayPos[3];         // ray origin
    float rayDir[3];         // ray direction vector
    float uvt[3]             // parametric solution
    float ctm[16];           // geometries attitude matrix
    float v0[4],n0[3],u0,v0; // explicit coordinates vertex 0
    float v1[4],n1[3],u1,v1; // explicit coordinates vertex 1
    float v2[4],n2[3],u2,v2; // explicit coordinates vertex 2
    float du1, dv1;           // texture direction vector 1
    float du2, dv2;           // texture direction vector 2
    float u,v;               // texture hit coordinate

    // first, calculate 3D picking ray from screen coordinates
    VRT_CalcPickingRay(x, y, rayPos, rayDir);

    // retrieve nodes attitude matrix
    VRT_NodeGetTransform(ntriangle,ctm);

    // read triangles vertex data
    VRT_GeometryGetVertex(gtriangle,0,&v0[0],&v0[1],&v0[2],
                          &n0[0],&n0[1],&n0[2],
                          &u0,&v0);
    VRT_GeometryGetVertex(gtriangle,1,&v1[0],&v1[1],&v1[2],
                          &n1[0],&n1[1],&n1[2],
                          &u1,&v1);
    VRT_GeometryGetVertex(gtriangle,2,&v2[0],&v2[1],&v2[2],
                          &n2[0],&n2[1],&n2[2],
                          &u2,&v2);

    // set homogeneous coordinate component
    v0[3] = 1.0f;
    v1[3] = 1.0f;
    v2[3] = 1.0f;

    // transform local coordinates into WCS
    VRT_TransformVertex3D(v0,ctm);
    VRT_TransformVertex3D(v1,ctm);
    VRT_TransformVertex3D(v2,ctm);

    // test for ray-triangle intersection
    if (VRT_IntersectRayTriangleUVT(rayPos,rayDir,v0,v1,v2,uvt))
```
// calculate first texture direction vector
du1 = u1-u0;
dv1 = v1-v0;

// calculate second texture direction vector
du2 = u2-u0;
dv2 = v2-v0;

// calculate texture hit position
u = u0 + uvt[0]*du1 + uvt[1]*du2;
v = v0 + uvt[0]*dv1 + uvt[1]*dv2;

//do something with the floating point texel hit coordinate
return;