A05 – Parallel projections

The Vulkan application whose source code is contained in file `A06.cpp`, shows a stub of an escape room game, where the user has to properly insert a key into a keyhole to unlock the door and exit. To complete the stub, you have to write two functions in file `WorldVies.hpp` to compute respectively the View-Projection matrix (considering the camera), and the World matrix for the Key.

```
glm::mat4 MakeViewProjectionMatrix(float Ar,
                                   float Alpha, float Beta, float Rho,
                                   glm::vec3 Pos)
```

- Creates a *View-Projection Matrix*, with near plane at *0.1*, and far plane at *50.0*, and aspect ratio given in `Ar`. The view matrix, uses the Look-in-Direction model, with vector `pos` specifying the position of the camera, and angles `Alpha`, `Beta` and `Rho` defining its direction. In particular, `Alpha` defines the direction (Yaw), `Beta` the elevation (Pitch), and `Rho` the roll.

glm::mat4 MakeWorldMatrix(glm::vec3 pos, glm::quat rQ, glm::vec3 size)

- creates and returns a *World Matrix* that positions the object at `pos`, orients it according to `rQ`, and scales it according to the sizes given in vector `size`.

You can move the view using the same keys as in *Assigment0*:

ESC – quit the application		SPACE BAR – switch between camera and key				
Q: roll CCW	**W**: forward	**E**: roll CCW	**R**: up		↑: look up	
A: left	**S**: backward	**D**: right	F: down	←: look left	↓: look down	→: look right

Once you have solved the assignment, please have a look at the following part of the other files of the assignment, since they can be considered as a reference for seeing the implementation of other topics considered in lessons 6 and 7.

Function `getSixAxis()` in file `Starter.hpp`, shows how to use GLFW functions to query the keyboard, the mouse and the joystick, and use them to compute six axes values (plus a fire button). The function also determines the time that has elapsed since the last call to properly synchronize motion with frame rate.

```
      void getSixAxis(float &deltaT, glm::vec3 &m, glm::vec3 &r, bool &fire) {
            static auto startTime = std::chrono::high_resolution_clock::now();
            static float lastTime = 0.0f;

            auto currentTime = std::chrono::high_resolution_clock::now();
            float time = std::chrono::duration<float,
                              std::chrono::seconds::period>
                             (currentTime - startTime).count();
            deltaT = time - lastTime;
            lastTime = time;
…
```

Function `updateUniformBuffer(…)` of fie `A06.cpp` has an example of implementing the walk model to control a camera, in the `true` section of the `if(MoveCam)` line.

```
if(MoveCam) {
      CamAlpha = CamAlpha - ROT_SPEED * deltaT * r.y;
      CamBeta  = CamBeta  - ROT_SPEED * deltaT * r.x;
      CamBeta  =  CamBeta < glm::radians(-90.0f) ? glm::radians(-90.0f) :
                 (CamBeta > glm::radians( 90.0f) ? glm::radians( 90.0f) :
                  CamBeta);
      CamRho   = CamRho   - ROT_SPEED * deltaT * r.z;
      CamRho   =  CamRho < glm::radians(-180.0f) ? glm::radians(-180.0f) :
                           (CamRho > glm::radians( 180.0f) ?
                           glm::radians( 180.0f) : CamRho);
```

the `else` section of the same `if(MoveCam)` statement, shows instead how to properly handle a quaternion based rotation of an object:

```
} else {
      KeyPos.x += MOVE_SPEED * m.x * deltaT;
      KeyPos.y += MOVE_SPEED * m.y * deltaT;
      KeyPos.z += -MOVE_SPEED * m.z * deltaT;
      KeyRot = glm::quat(glm::vec3(0, -ROT_SPEED * deltaT * r.y, 0)) *
               glm::quat(glm::vec3(-ROT_SPEED * deltaT * r.x, 0, 0)) *
               glm::quat(glm::vec3(0, 0, ROT_SPEED * deltaT * r.z)) *
                         KeyRot;
}
```