Appendices

Appendix A: Geometry Classes

This section gives a description of the methods in `Point3`, `Vec3`, `Triangle` and `Matrix` classes. The static relationships between the classes are shown in Fig. A.1.

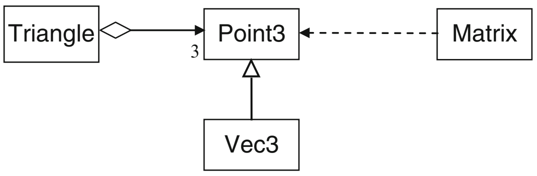

Fig. A.1 Relationships between geometry classes

A.1 Point3 Class

Fields

```
protected:
      static float EPS;
public:
      float _x, _y, _z, _h;
```

Description:

The data members of the class store the coordinates of a point. For programming convenience, the coordinates are declared as `public`, so that they can be directly accessed without the need for getter methods. The fourth component `_h` is initialized to 1 for points and 0 for vectors. This component is not used for computing the norm, scalar product, and other operations such as addition, subtraction and negation.

The static field `EPS` is a threshold used for checking if a floating point value is close enough to zero. Its value is set to 1.E-6.

Constructors

```
public:
      Point3(float x, float y, float z = 0)
            : _x(x), _y(y), _z(z), _h(1.0)  {}
      Point3()
            : _x(0.0), _y(0.0), _z(0.0), _h(1.0)  {}
```

Description:

The first constructor sets the values of *x*, *y*, *z* coordinates using its arguments. The *h* value is initialized separately to a default value 1.0. The second no-argument constructor initializes a point to the origin.

Distance computation

```
float norm() const;
```

Description:

This method computes the distance to a point from the origin or the length of a vector.

Addition and subtraction

```
Point3* add(const Point3* p) const;
Vec3* subtract(const Point3* p) const;
```

Description:

The `add` method adds the *x*, *y*, *z* coordinates of the current point with the corresponding coordinate values of `p`, and produces a new point. The *h* coordinate values are not added. The resulting point is assigned an *h* value 1.0. This method is overridden in the subclass `Vec3` which sets the *h* value to 0. The `subtract` method similarly subtracts the coordinates of `p` from that of the current point and produces a vector originating at `p`.

Negation

```
Point3* negate() const;
```

Description:

This method negates the *x*, *y*, *z* coordinates of the current point. The *h* coordinate value is not negated.

Scalar multiplication

```
Point3* scalarMult(float c) const;
```

Description:

This method scales the *x*, *y*, *z* coordinates of the current point by the constant factor `c`, and produces a new point. The resulting point is assigned an *h* value 1.0.

Conversion to standard form

```
Point3* standard();
```

Description:

This method converts the current point to standard form by applying the transformation: $(x, y, z, h) \Rightarrow (x/h, y/h, z/h, 1)$, provided $h \neq 0$.

Output

```
void print() const;
```

Description:

This method prints the x, y, z, h coordinates of the current point or vector.

A.2 Vec3 Class

The `Vec3` class is a subclass of `Point3`.

Fields

```
private:
      static float RADTODEG;
public:
      static const Vec3* X_AXIS;
      static const Vec3* Y_AXIS;
      static const Vec3* Z_AXIS;
```

Description:

The static field `RADTODEG` stores the multiplication factor ($= \pi/180$) for conversion from radians to degrees.

The static fields `X_AXIS`, `Y_AXIS`, `Z_AXIS` store respectively the orthogonal basis vectors (1, 0, 0), (0, 1, 0) and (0, 0, 1).

Constructors

```
public:
      Vec3(float x, float y, float z = 0)
           : Point3(x, y, z){_h = 0;}
      Vec3() {_h = 0;}
```

Description:

The constructors invoke the base class constructors and additionally set the value of `_h` to 0.

Dot and cross products

```
float dot(const Vec3* v) const;
Vec3* cross(const Vec3* v) const;
```

Description:

The `dot` method returns the dot product of the current vector and `v`. The `cross` method returns a vector as the result of the cross product between the current vector and `v`.

Vector normalization

```
        void normalize();
```

Description:

The method converts the current vector to a unit vector by dividing its components by the length of the vector.

Reflection of a vector

```
        Vec3* reflect(const Vec3* n) const;
```

Description:

The method computes the reflection of the current vector with respect to $\boldsymbol{n}$ using the formula in Eq. 2.5.

Computation of angles

```
        float angle(const Vec3* v) const;
        float angle2(const Vec3* v) const;
        float signedAngle(const Vec3* v, const Vec3* w) const;
```

Description:

The method `angle` first converts the current vector and the input vector `v` to unit vectors, and then computes the angle between them using the inverse cosine of the dot product of the two vectors. The value is returned in degrees in the range [0, 180]. The method `angle2` uses both dot and cross products to compute the angle using the formula $\theta = \tan^{-1}(|\boldsymbol{u} \times \boldsymbol{v}|,\ \boldsymbol{u} \bullet v)$. The `singedAngle` method uses Eq. 2.6. to compute the signed angle between the current vector and `v` with respect to a given view direction `w`.

A.3 Triangle Class

Fields

```
 private:
        const Point3 *_a, *_b, *_c;
```

Description:

The data members of the class store references to the three vertices of a triangle.

Constructors

```
 public:
     Triangle(const Point3* a, const Point3* b, const Point3* c)
             : _a(a), _b(b), _c(c)  {}
```

Description:

The non-default constructor requires three references to objects of the `Point3` class. Methods of the class use these points as vertices of the triangle.

Computation of area

```
float area() const;
float signedArea2D() const;
float signedArea3D(const Vec3* w) const;
```

Description:

The method `area` computes the area of the current triangle using the cross product of vectors along two edges as given in Eq. 2.3. The method `signedArea2D` returns the area of the triangle which has a negative sign if the angle between the normal direction and the z-axis is greater than 90°. The function `signedArea3D` uses a similar approach by using a user specified vector $\boldsymbol{w}$ instead of the z-axis (Eq. 2.8).

Computation of barycentric coordinates

```
Point3* barycentricCoords(const Point3* p) const;
```

Description:

This method computes the barycentric coordinates of the point `p` with respect to the current triangle using area ratios as given in Eq. 2.48.

Barycentric mapping

```
Point3* barycentricMap
 (const Point3* p, const Triangle* t) const;
```

Description:

A point `p` and a triangle `t` containing `p` are given. This method computes the image of `p` in the current triangle as shown in Fig. 2.12.

Point inclusion test

```
bool isInside(const Point3* p) const
```

Description:

This function uses barycentric coordinates to determine if a point `p` lies within and on the plane of the current triangle.

Bilinear interpolation

```
Point3* bilinear(int k1, int k2) const
```

Description:

This method returns a point computed using the bilinear interpolation formula in Eq. 2.45. The arguments k_1 and k_2 must satisfy the condition that k_1, k_2, and $k_1 + k_2$, all have values in the range [0, 1].

OpenGL drawing

```
void draw() const;
```

Description:

This method draws the current triangle using OpenGL functions.

A.4 Matrix Class

Fields

```
private:
      float _v[4][4];
```

Description:

The `Matrix` class represents the data structure for a 4 × 4 matrix, with its values stored in the two-dimensional array `_v`.

Constructors

```
public:
      Matrix()
      Matrix(float values[][4])
      Matrix(const Vec3* u, const Vec3* v, const Vec3* w)
```

Description:

The default constructor initializes the matrix with the identity matrix. The second constructor initializes the matrix using a two-dimensional array of values. The values are stored in row-major order. The third constructor forms the matrix using three vectors $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ as the first three columns of the matrix. The last column has values 0, 0, 0, 1.

Identity matrix

```
      void identity();
```

Description:

This method resets the current matrix to the identity matrix.

Accessing matrix elements

```
      float valueAt(int i, int j) const;
```

Description:

This is a getter method that returns the value of `_v[i][j]`.

Setting matrix elements

```
      void setValue(int i, int j, int value);
```

Description:

This is a setter method that replaces the value of `_v[i][j]` with `value`.

Transpose and inverse

```
      void transpose();
      void inverse();
```

Description:

The method `transpose` modifies the current matrix by replacing it with its transpose. Similarly `inverse` replaces the current matrix with its inverse, provided the matrix is invertible. If the determinant of the current matrix is 0, it is not changed.

Point transformation

```
        Point3* transform(const Point3* p);
```

Description:

This method returns a new point computed by pre-multiplying the point `p` by the current matrix.

Matrix copy

```
        Matrix* copy();
```

Description:

Often it is required to keep a copy of the current matrix before computing its transpose or inverse. This method returns a reference to a new matrix object that contains the same values as the current matrix.

Output

```
    void print();
```

Description:

This method prints the values of the current matrix in 4×4 format.

Appendix B: Scene Graph Classes

This section gives an outline of the methods in the scene graph classes. A description of these classes can be found in Sect. 3.5. The static relationships between the classes are shown in Fig. B.1.

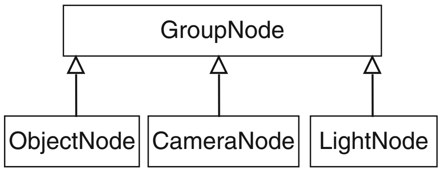

Fig. B.1 Relationships between scene graph classes

B.1 GroupNode Class

Fields

```
private:
      list<GroupNode*> _children;
protected:
      GroupNode* _parent;
      float _tx, _ty, _tz, _angleX, _angleY, _angleZ;
```

Description:

The list variable `_children` stores references to the children of the current group node, in an STL list structure. The access level for this variable is declared as `private` since all subclasses are leaf nodes that do not have children. Each group node also stores a reference to its parent node in the variable `_parent`. It has a value `NULL` for the root node. Every group node also stores the translation parameters `_tx`, `_ty`, `_tz` and rotation angles `_angleX`, `_angleY`, `_angleZ` which define the transformation of the current node to the coordinate frame of the parent node.

Constructors

```
GroupNode()
```

Description:

The class contains only one no-argument constructor that initializes the parent node to `NULL` and the transformation parameters to zeros.

Add/remove child

```
void addChild(GroupNode* node);
void removeChild(GroupNode* node);
```

Description:

The method `addChild` includes the specified `node` as a child node of the current node. The method `removeChild` removes the specified `node`, if it exists, from the list of children of the current node.

Node transformation

```
void translate(float tx, float ty, float tz);
void rotateX(float angle);
void rotateY(float angle);
void rotateZ(float angle);
```

Description:

The above methods set the transformation parameters of the current node. The node transformation is always assumed to be of the form **TR**.

Inverse transformation

```
void inverseTransform() const;
```

Description:

This method uses OpenGL functions to push the matrices for the inverse transformation $(\mathbf{TR})^{-1} = \mathbf{R}^{-1}\mathbf{T}^{-1}$ of the current node to the transformation stack. Note that this function does not explicitly generate the inverse transformation matrix.

Scene rendering

```
 void render();
virtual void draw();
```

Description:

This method gets the singleton object of the `CameraNode`, sets up the view transformation matrix and calls the method `draw`. A scene is rendered by invoking this method on the root node.The `draw` method is not directly invoked by the application. It is indirectly invoked on a group node via the method `render`. The `draw` method uses OpenGL functions to push the current node's transformation matrix to the transformation stack, and recursively calls itself on all child nodes. This polymorphic method causes objects to be drawn when invoked on leaf nodes.

Parent node

```
      GroupNode* getParent() const;
```

Description:

This getter method returns the reference to the parent of the current node.

B.2 ObjectNode Class

The `ObjectNode` class is a subclass of `GroupNode`.

Fields

```
public:
    enum ObjType
      { CUBE, SPHERE, TORUS, TEAPOT, CONE, TETRAHEDRON};
private:
      ObjType _object;
      float _scaleX, _scaleY, _scaleZ;
      float _colorR, _colorG, _colorB;
```

Description:

The enumerated type `ObjType` defines a collection of GLUT objects which users can specify in the constructor to display an object. At the time of construction, the user can specify its scale factors `_scaleX`, `_scaleY`, `_scaleZ`, and also its material colour using the normalized values in the range [0, 1] for `_colorR`, `colorG`, `colorB`.

Constructors

```
public:
      ObjectNode()      : GroupNode(),   _object(CUBE),
             _scaleX(1.0f), _scaleY(1.0f), _scaleZ(1.0f),
             _colorR(1.0f), _colorG(1.0f), _colorB(1.0f)  {}
```

Description:

The constructor initializes the object type to CUBE, the scale factors to 1, and the object material colour to white.

Setter methods

```
      void setObject(ObjType object,
           float scaleX, float scaleY, float scaleZ);
      void setColor(float colorR, float colorG, float colorB);
```

Description:

The method `setObject` is used to change the parameters of the current object, including its type and scale factors. The `setColor` method modifies the material colour of the current object.

B.3 CameraNode Class

The `CameraNode` class is a subclass of `GroupNode`.

Fields

```
private:
      float _fov, _aspect, _near, _far;
      static bool flag;
```

Description:

The data members `_fov`, `_aspect`, `_near`, `_far` define the perspective view frustum of the camera in terms of the field of view, aspect ratio, near plane distance and the far plane distance. The Boolean variable `flag` ensures that at most one instance of the class is created.

Constructors

```
private:
     CameraNode(): GroupNode(), _fov(60.0f), _aspect(1.0f),
                         _near(1.0f), _far(1000.0f) {}
```

Description:

The `CameraNode` class is a singleton class with a `private` constructor. The only instance of the class is available through the static method `getInstance()`. By default, the camera view frustum has 60° field of view, aspect ratio 1, near plane distance 1, and far plane distance 1,000.

Setter method

```
      void perspective
           (float fov, float aspect, float near, float far);
```

Description:

This setter method allows you to change the default frustum parameters of the camera object.

View transformation and projection

```
      void viewTransform() const;
      void projection() const;
```

Description:

The `viewTransform` method traverses the scene graph from the camera node towards the root node, and pushes the inverse transformation matrices of each node onto the transformation stack using OpenGL functions. The method calls the `inverseTransform` method of the `GroupNode` class for this operation.

The method `projection` sets up the projection matrix using OpenGL functions. Both the above methods are not usually invoked directly by the user. The `render` method of the `GroupNode` class invokes both the methods to set up the view and projective transformations for the rendering pipeline.

B.4 LightNode Class

The `LightNode` class is a subclass of `GroupNode`.

Fields

```
private:
      int glLight;
```

Description:

This integer field can be assigned a value between 0 and 7. A value *i* corresponds to the named light source GL_LIGHT*i* defined in OpenGL.

Constructors

```
public:
      LightNode(int glLight): GroupNode(),_glLight(glLight) {}
```

Description:

The constructor specifies the index of the OpenGL light source to be used for the current object of the `LightNode` class. The default position of the light node is (0, 0, 0). The position can be changed by specifying transformation parameters for the node using the `translate` method. Note that all other light source parameters will have to be defined separately by the user with the help of OpenGL functions.

Setter Method

```
      void setLight(int glLight);
```

Description:

This setter method allows the user to change the current light source used by the object.

Appendix C: Vertex Skinning Classes

This section gives an outline of the methods in the `SkinnedMesh`, `Skeleton` and `SkeletonNode` classes used for vertex skinning. A description of these classes can be found in Sect. 4.8. A class diagram showing the relationships between the classes is given in Fig. C.1.

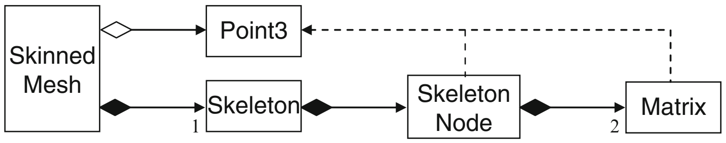

Fig. C.1 Relationships between the classes used for vertex skinning

C.1 SkeletonNode Class

The structure of the `SkeletonNode` class is similar to that of the `GroupNode` class.

Fields

```
private:
      list<SkeletonNode*> _children;
      int _firstIndex, _lastIndex, _parentIndex;
      SkeletonNode* _parent;
      float _tx, _ty, _tz, _angleX, _angleY, _angleZ;
      Matrix *_matrix, *_invMatrix;
```

Description:

Every skeleton node is implicitly a group node, and can store references to a number of children in the list `_children`. A skeleton node represents a bone. It also stores a pair of indices `_firstIndex` and _lastIndex defining a range of mesh vertices that are attached to the bone. The translation parameters are stored in variables _tx, _ty, _tz, and the Euler angles in _angleX, _angleY, _angleZ. The overall transformation matrix and its inverse are updated whenever any of the joint angles is changed. Each node is assigned a unique index starting form 1. The index 0 is reserved for the root node which represents the origin of the world coordinate system. The parent index _parentIndex establishes the link between the current node and its parent node.

Constructors

```
public:
    SkeletonNode(int parentIndx,
        float tx, float ty, float tz,
        int firstIndx, int lastIndx)
            : _parent(NULL),
              _tx(tx), _ty(ty), _tz(tz),
              _angleX(0.0), _angleY(0.0), _angleZ(0.0),
              _firstIndex(firstIndx), _lastIndex(lastIndx),
              _parentIndex(parentIndx)
            {     _matrix = new Matrix();
                  _invMatrix = new Matrix();
                  updateMatrices();   }
```

Description:

The non-default constructor uses the parameters read in from the input file to initialise each node. Note that each node contains two instances of the matrix class. Both the transformation matrix and its inverse are updated using the input parameters. There is also a default constructor that initializes all transformation parameters to 0.

Add/remove child:

```
        void addChild(SkeletonNode* node);
        void removeChild(SkeletonNode* node);
```

Description:

These methods are exactly the same as the corresponding methods in the GroupNode class (Appendix B).

Bone transformations: translation

```
        void translate(float tx, float ty, float tz);
```

Description:

The translation parameters of the bones are set at the time of construction, and do not change afterwards. Only the translation of the base node (with respect to the world coordinate frame) is defined in the animation phase. This method is therefore usually invoked by the translateBase method of the Skeleton class.

Bone transformations: rotation

```
void rotateX(float angle);
void rotateY(float angle);
void rotateZ(float angle);
```

Description:

These methods are used to set the rotation angle(s) of a bone during the animation phase. The methods are normally invoked by the `rotate` method of the `Skeleton` class.

Setter methods

```
void attachVertices(int firstIndx, int lastIndx);
void setParentIndex(int parentIndx);
```

Description:

These methods alter the vertex indices and the parent index of the current bone.

Getter methods

```
int getParentIndex() const;
int getFirstIndex() const;
int getLastIndex() const;
Matrix* getMatrix() const;
Matrix* getInverseMatrix() const;
SkeletonNode* getParent() const;
int getChildCount() const;
```

Description:

These methods allow you to examine the transformation matrices, vertex indices, the parent index and the number of children of the current node.

Transformation update

```
void updateMatrices();
```

Description:

This method updates the transformation matrix and its inverse, and is invoked whenever any of the transformation parameters is changed.

Pre-processing phase

```
vector<Point3*> preprocessPhase(vector<Point3*> vertices);
void transform1
 (vector<Point3*> vertices, float tx, float ty, float tz);
```

Description:

The pre-processing phase builds the product matrix given in Eq. 4.9 and transforms the mesh vertex list to create a new list of vertices V'. The method `preprocessPhase` returns this new vertex list. The method in turn invokes `transform1` which traverses the skeleton tree from the root, visits every node, combines the inverse translation components, and applies the transformation on the node's vertex list.

Animation phase

```
        vector<Point3*> animationPhase(vector<Point3*> vertices);
        void transform2(vector<Point3*> vertices, Matrix matrix);
```

Description:

In the animation phase, the updated matrices incorporating joint angle rotations are gathered in the form of a product matrix given in Eq. 4.10. The vertex list obtained from the pre-processing phase is transformed using the matrix. The transformed vertex coordinates returned by `animationPhase` are used for rendering the mesh for that particular frame. This method invokes `transform2` which traverses the skeleton tree from the root, post-multiplies the product matrix with the matrix at the current node, and transforms the node's vertices obtained from the pre-processing phase.

C.2 Skeleton Class

Fields

```
        private:
              SkeletonNode* _root;
              SkeletonNode* _base;
              vector<SkeletonNode*> _bones;
```

Description:

Each skeleton tree is referenced by its root node, stored in the variable `_root`. This node is created by the constructor. The base node (`_base`) is a special node in the skeleton tree that has the root node as its parent. The transformations of the base node define the position and the orientation of the entire mesh in the world coordinate frame. The class also maintains a list of references to the skeleton nodes as they are created by the `loadSkeleton` method.

Constructors

```
public:
      Skeleton()
        : _root( new SkeletonNode() )  {}
```

Description:

The constructor creates the root node of the skeleton and initializes it with the default transformation parameters.

Getter method

```
        SkeletonNode* getRoot() const;
```

Description:

The getter method returns the reference to the root node.

Loading skeleton data

```
      void loadSkeleton(const string& filename);
```

Description:

This method loads skeleton data from a file formatted as shown in Fig. 4.18, and creates an instance of the `SkeletonNode` class for each bone. The method also calls `attachBones` that creates the hierarchical relationships between nodes (bones).

Bone transformations

```
      void rotate(int i,
              float angleX, float angleY, float angleZ);
      void translateBase(float tx, float ty, float tz);
```

Description:

The translation parameters specifying the spatial offsets of each bone relative to its parent are assigned to the nodes through the constructor. These parameters are used for transforming vertices in the pre-processing phase. In the animation phase, only the joint angle rotations and the translation of the base node can change. The `rotate` method specifies the joint angles of the *i*th bone. The `translateBase` method changes the translation parameters of the base node. These two methods are usually called within the display loop of the application.

C.3 SkinnedMesh Class

Fields

```
private:
      vector<Point3*> _verticesV;
      vector<Point3*> _verticesVT;
      vector<Point3*> _verticesW;
      vector<Polygon*> _polygons;
      PolyType _polytype;
      float _colorR, _colorG, _colorB;
      Skeleton* _skeleton;
public:
    enum PolyType {TRIANGLE, QUAD};
```

Description: The vertex lists `verticesV`, `_verticesVT`, `_verticesW` represent the lists V, V', W shown in Fig. 4.11. The lists contain the mesh coordinates in the bind pose, after the pre-processing phase, and after the animation phase respectively. The polygon list `_polygons` store the vertex indices of the mesh polygons. For the sake of simplicity, each mesh is assigned a single material colour given by `_colorR`, `_colorG`, `_colorB`. A mesh can be either a triangular or a quad mesh. The variable `_skeleton` stores the reference to the skeleton associated with the mesh.

Constructors

```
public:
      SkinnedMesh(PolyType polytype)
            : _polytype(polytype), _skeleton(NULL)  {}
```

Description:

The constructor specifies only the polygon type of the mesh using the enumerated types `TRIANGLE`, and `QUAD`. Mesh data is loaded using `loadMesh` method. The application must also load skeleton data using an instance of the `Skeleton` class, and attach the skeleton object using the `attachSkeleton` method.

Loading a mesh

```
      void loadMesh(const string& filename);
```

Description:

This method loads mesh data from a file formatted as shown in Fig. 4.19. The number of vertices per polygon in the file should match the polygon type provided to the constructor. The method also populates the vertex lists `_verticesV` and `_verticesW` with the initial vertex coordinates obtained from the file. The polygon list `_polygons` is also populated with polygon data.

Getter method

```
      Skeleton* getSkeleton() const;
```

Description:

The getter method returns the reference to the skeleton object attached to the current `SkinnedMesh` object.

Setting mesh colour

```
      void setColor(float colorR, float colorG, float colorB);
```

Description:

The method sets a material colour for the entire mesh.

Attaching a skeleton

```
      void attachSkeleton(Skeleton* skeleton);
```

Description:

This method associates a skeleton object with the current mesh. The pre-processing of mesh vertices V to obtain an intermediate set of vertices V' (Eq. 4.9) is also initiated at this stage.

Rendering a mesh

```
void render();
```

Description:

This method is usually called inside the display loop of the application for redrawing the mesh with the updated joint angle configuration. Typically, this method is called after specifying the bone transformations using the `rotate` method of the `Skeleton` class.

Appendix D: Quaternion Classes

This section gives an outline of methods in the classes that represent quaternion and dual quaternion numbers. Figure D.1 shows the static relationships between the classes and the geometry classes.

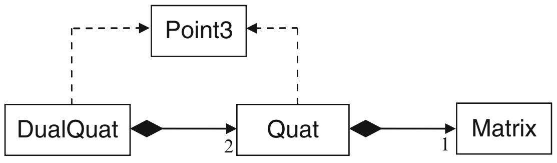

Fig. D.1 Relationships between the quaternion classes and the geometry classes

D.1 Quaternion Class

Fields

```
private:
      Matrix* _mat;
      static float RADTODEG;
      static float DEGTORAD;
      static float EPS;
public:
      float _q0, _q1, _q2, _q3;
```

Description:

Every quaternion object has an associated 4×4 transformation matrix `_mat`. The matrix elements are not automatically updated. The user needs to call `updateMatrix` to compute the values of the matrix elements. The constants `RADTODEG` and `DEGTORAD` store the conversion factors from radians to degrees and degrees to radians respectively. The quaternion components `_q0`, `_q1`, `_q2`, `_q3` are declared as public as they are frequently accessed. `EPS` stores the constant value 1.E-6 used as a threshold for checking if a value is close to zero.

Constructors

```
public:
      Quat(float q0, float q1, float q2, float q3);
      Quat(const Point3* p);
      Quat(float angle, const Vec3* axis);
      Quat()
```

Description:

The first constructor initializes an object with four quaternion components. The second constructor takes a point $P = (x, y, z)$ as the argument, and forms the pure quaternion $(0, x, y, z)$. The third constructor forms a unit quaternion using the angle and axis of a three-dimensional rotation as parameters. The quaternion is constructed as per Eq. 5.44. The fourth no-argument constructor initializes the quaternion components to (1, 0, 0, 0).

Getter methods

```
      Matrix* getMatrix() const;
      Point3* getPoint() const;
      float getAngle() const;
      Vec3* getAxis() const;
      Vec3* getEuler() const;
```

Description:

The first getter method given above returns the current matrix `_mat`. The second getter method returns the last three components `_q1`, `_q2`, `_q3` of the current quaternion as a point. The third and fourth getter methods return respectively the angle and axis of the equivalent rotation given by Eqs. 5.45 and 5.46. The method `getEuler` extracts the Euler angles from the quaternion components using Eq. 5.56.

Quaternion operations

```
      Quat* add(const Quat* q) const;
      Quat* subtract(const Quat* q) const;
      Quat* mult(const Quat* q) const;
      Quat* scalarMult(float term) const;
      Quat* conjugate() const;
      Quat* negate() const;
```

Description:

The methods listed above perform algebraic operations of addition, subtraction, multiplication, scalar multiplication, conjugation and negation, and return the resulting quaternion.

Quaternion norm

```
float norm() const;
```

Description:

The above method returns the magnitude of the current quaternion (Eq. 5.17).

Quaternion matrix

```
void updateMatrix();
```

Description:

Each quaternion object has an associated transformation matrix as given in Eq. 5.23. The above method must be called whenever a quaternion component has changed, in order to update this matrix.

Quaternion transformation

```
Point3* transform(const Point3* point);
```

Description:

The above method transforms a point using the current quaternion according to the formula $P' = QPQ^*$.

Conversion to unit quaternion

```
void normalize();
```

Description:

The method `normalize` converts the current quaternion to a unit quaternion.

Quaternion interpolation

```
Quat* lerp(float t, Quat* q);
Quat* slerp(float t, Quat* q);
```

Description:

The above methods perform linear (`lerp`) and spherical linear (`slerp`) interpolations between the current quaternion and the supplied quaternion `q`, and return an intermediate quaternion for the parameter value given by `t`.

Output

```
void print();
```

Description:

The above method prints the component values of the current quaternion.

D.2 Dual Quaternion Class

Fields

```
private:
Quaternion *_quat1, *_quat2;
```

Description:

Each dual quaternion is composed using two quaternions _quat1, _quat2 as described in Sect. 5.9.2.

Constructors

```
DualQuat(const Quat* quat1, const Quat* quat2);
DualQuat(float angle, const Vec3* axis, const Vec3* trans);
DualQuat(const Point3* p);
```

Description:

The first constructor shown above forms a dual quaternion using two quaternion components. The second constructor using the rigid-body transformation parameters (angle and axis of rotation, and translation vector) to construct the equivalent dual quaternion. The third constructor creates the dual quaternion $(1, 0, 0, 0, 0, x, y, z)$ using the coordinates (x, y, z) of the specified point.

Getter methods

```
Quat* getQuat1() const;
Quat* getQuat2() const;
Point3* getPoint() const;
```

Description:

The first two methods shown above return respectively the first and the second quaternion components of the current dual quaternion. The third method returns the last three elements (of the second quaternion component) as the coordinates of a point.

Product of two dual quaternions

```
DualQuat* mult(const DualQuat* q) const;
```

Description:

The above method returns the product of the current dual quaternion and the specified dual quaternion (`q`). The product is computed using the formula in Eq. 5.85.

Product of a dual quaternion and a quaternion

```
DualQuat* multQuat(const Quat* q) const;
```

Description:

The above method returns the product of the current dual quaternion and the specified quaternion (`q`). The product is computed using the formula in Eq. 5.86.

Dual quaternion transformation

```
      Point3* transform(const Point3* point);
```

Description:

The above method transforms a point using the current quaternion according to the formula in Eq. 5.97.

Output

```
      void print();
```

Description:

The above method prints the component values of the current dual quaternion.

Index