РОССИЙСКАЯ АКАДЕМИЯ НАУК
СИБИРСКОЕ ОТДЕЛЕНИЕ
А В Т О М Е Т Р И Я
№ 2 1994

УДК 681.3.06

А. В. Романовский
(Новосибирск)

ИСПОЛЬЗОВАНИЕ ТИПОВ ДАННЫХ ДЛЯ ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ ДИНАМИКИ ТРЕХМЕРНЫХ СЦЕН

Применение механизма расширения типов данных позволяет построить иерархию программных моделей динамики трехмерных сцен, в которой более сложные модели используют частично или полностью программный код для более простых моделей. Такой подход дает возможность значительно уменьшить затраты на проектирование, создание, сопровождение и развитие программ, описывающих визуальные модели динамики трехмерных сцен.

Введение. Для решения задач машинной графики в последнее время стал использоваться объектно-ориентированный подход [1]. Главным фактором распространения этого подхода является уменьшение с его помощью затрат на проектирование, создание, сопровождение и развитие программ.

В данной статье приведен простой набор типов, традиционно используемых в машинной графике, и показано, как последовательное применение принципов объектно-ориентированного программирования позволяет строить и усложнять программные модели динамики трехмерных сцен.

Для формального определения типов данных и написания примеров в работе используется язык программирования высокого уровня Oberon-2 [2].

Например, определение модуля, реализующего тип LIFO-список, на языке Oberon-2 может выглядеть так:

```
MODULE LIFO;
TYPE
    NodePointer* = POINTER TO Node;
    Node* = RECORD previous- : NodePointer; END;
    List* = RECORD last- : NodePointer; END;

PROCEDURE (VAR list : List) Default* ( );
(*Конструктор пустого списка*)
    BEGIN list.last: = NIL; END Default;

PROCEDURE (VAR l : List) Top* (n : NodePointer);
(*Добавление узла в список*)
    BEGIN n^.previous: = l.last; l.last: = n; END Top;

PROCEDURE (VAR l : List) Untop* ( );
(*Удаление последнего узла из списка*)
    BEGIN
        IF l.last = NIL THEN (*Обработка ошибки*)
        ELSE
            l.last: = l.last^.previous;
        END;
    END Untop;
END LIFO;
```

Геометрические типы данных. Типы данных, необходимые для описания геометрии в 3-мерном пространстве, определены в модуле XYZ. Это типы:

— Vector, объекты данных которого задают векторы в 3-мерном пространстве;

— Plane, объекты данных которого задают плоскости в 3-мерном пространстве;

— Matrix, объекты которого задают преобразования координат в 3-мерном пространстве.

```
MODULE XYZ;
TYPE
    Vector* = ARRAY 3 OF REAL;
    Plane* = RECORD normal- : Vector; distance_to_origin- : REAL; END;
    Matrix* = ARRAY 4 OF Vector;
VAR
    ortX-,ortY-,ortZ-,origin- : Vector;

PROCEDURE NewVector* (VAR result : Vector; x,y,z : REAL);
(*Конструктор вектора*)
    END NewVector;

PROCEDURE AddVectors* (VAR result : Vector; a,b : Vector);
(*Сложение векторов a, b*)
    END AddVectors;

PROCEDURE Normalize*(VAR v : Vector);
(*Нормализация вектора. Если длина вектора не равна 0, то после вызова
данной процедуры она будет равна 1*)
    END Normalize;

PROCEDURE NewMatrix*(VAR result : Matrix; x,y,z,o : Vector);
(*Конструктор преобразования координат*)
    END NewMatrix;

PROCEDURE XtoY*(VAR result : Matrix; angle : REAL);
(* Конструктор преобразования поворота от оси OX к OY на угол angle,
задаваемый в радианах *)
    END XtoY;

PROCEDURE transformVector*(VAR result : Vector;
                                    source : Vector; t : Matrix);
(*source преобразуется в result в соответствии со значением t*)
    END transformVector;

PROCEDURE Compose*(VAR result : Matrix; a,b : Matrix);
(*В result возвращается результат композиции преобразований a и b*)
    END Compose;

BEGIN
    NewVector(ortX,1,0,0); NewVector(ortY,0,1,0);
    NewVector(ortZ,0,0,1); NewVector(origin,0,0,0);
END XYZ.
```

«Проволочная» модель 3-мерных поверхностей. Для программного описания «проволочной» модели используются типы:

— Vertex, объекты данных которого задают отдельные вершины граней;

— VertexList, объекты данных которого задают списки вершин граней;

— Face, объекты данных которого задают грани поверхности в «проволочной» модели; вершины каждой грани представляются массивом указателей на отдельные вершины, что позволяет иметь общие вершины в различных гранях;
— FaceList, объекты данных которого задают списки граней;
— Faces, объекты данных которого представляют наборы граней моделируемой поверхности.

Эти типы определены в модуле WIRE_FACES.

```
MODULE WIRE_FACES;

IMPORT LIFO, XYZ;

TYPE
    Vertex* = POINTER TO VertexData;
    VertexData* = RECORD(LIFO.Node)
        xyz* : XYZ.Vector;
    END;
    Vertices* = POINTER TO ARRAY OF Vertex;

    Face* = POINTER TO FaceData;
    FaceData* = RECORD(LIFO.Node)
        vertices- : Vertices;
        vertices_total- : INTEGER;
        plane : XYZ.Plane;
    END;

    FaceList* = RECORD(LIFO.List) END;
    VertexList* = RECORD(LIFO.List) END;

    Faces* = RECORD
        faces : FaceList;
        vertex_table : VertexList;
    END;

PROCEDURE NewVertex(v : XYZ.Vector) : Vertex;
(*Конструктор вершины*)
    END NewVertex;

PROCEDURE (VAR ff : Faces) Convex* (vv : VertexList);
(*Конструктор поверхности минимальной выпуклой оболочки списка вершин*)
    END Convex;

END WIRE_FACES.
```

Определение типов VertexData, FaceData, VertexList, FaceList как расширений соответствующих типов из модуля LIFO позволяет применять к вершинам, граням и их спискам процедуры из модуля LIFO без их изменения.

Модель 3-мерных поверхностей с однотонной закраской. Эта модель [3, с. 112] определяется как расширение полигональной «проволочной» модели. Для ее программного описания используются типы из модуля FLAT_FACES:
— Color, объекты которого задают цвета по 3-компонентной RGB-модели;
— Face, объекты которого задают однотонно окрашенные грани.

```
MODULE FLAT_FACES;
IMPORT W: = WIRE_FACES;
TYPE
    Color* = POINTER TO ColorData;
```

```
    ColorData* = RECORD red-,green-,blue- : REAL; END;
    Faces* = POINTER TO FaceData;
    FaceData* = RECORD(W.FaceData) color- : Color; END;
    Faces* = RECORD(W.Faces) END;

PROCEDURE (VAR ff : Faces) ColorBy* (c : Color);
(*Окраска граней сконструированной поверхности в один цвет*)
    END ColorBy;

END FLAT_FACES.
```

Следует отметить, что объекты типа FLAT_FACES.Faces могут быть созданы с помощью процедур-конструкторов из модуля WIRE_FACES, в частности посредством вызова процедуры Convex. Полученный таким образом набор граней может быть обработан любой процедурой из модуля FLAT_FACES, например процедурой ColorBy, которая позволяет «окрасить» грани в заданный цвет.

Полигональная модель с интерполяцией освещенности по методу Гуро. Эта модель [4] определяется как расширение предыдущих моделей в модуле GOURAUD_FACES.

```
MODULE GOURAUD_FACES;
IMPORT WIRE: = WIRE_FACES, FLAT: = FLAT_FACES, XYZ;
TYPE
    Vertex* = POINTER TO Vertex;
    VertexData* = RECORD(WIRE.VertexData) normal- : XYZ.Vector;
    END;
    Faces* = RECORD(FLAT.Faces) END;

PROCEDURE (VAR ff : Faces) Smooth* ();
(*Расстановка нормалей в общих вершинах граней для создания эффекта
визуальной гладкости*)
    END Smooth;
END GOURAUD_FACES.
```

Так же как и в предыдущей модели, объект типа GOURAUD_FACES.Faces может быть сконструирован и обработан при помощи процедур из модуля WIRE_FACES или FLAT_FACES, а затем полученный набор граней может быть обработан любой процедурой из модуля GOURAUD_FACES, например процедурой Smooth, которая позволяет расставить нормали в общих вершинах граней для создания эффекта визуальной гладкости моделируемой полигональной поверхности.

Системы координат. Размещение моделей полигональных объектов в сцене осуществляется с использованием систем, или пространств, координат. Считается, что вся моделируемая сцена находится в абсолютной или мировой системе координат. Вершины граней определяются в той системе координат, которая наиболее удобна для конструирования поверхности или тела. Такая система координат называется собственной системой координат объекта. Часто мировая система координат и система координат объекта не совпадают.

Существует несколько подходов для представления отношений между системами координат. Так, например, одна из систем координат может быть выбрана в качестве абсолютной системы координат. Все остальные системы координат, никак не связанные друг с другом, будут находиться внутри абсолютной системы координат, т. е. все системы координат образуют двухуровневую иерархию: на верхнем уровне находится абсолютная система координат, на нижнем уровне — все остальные системы координат.

При другом подходе системы координат образуют многоуровневую древовидную иерархию, в которой каждая система координат $n + 1$-го уровня считается вложенной в систему координат n-го уровня. Системы координат

n-го уровня называются базисными для систем координат уровня $n + 1$, а те, в свою очередь, называются подчиненными. У нескольких подчиненных систем координат может быть одна и та же базисная система координат. Перемещение базисной системы координат означает перемещение всех подчиненных систем координат. Связи между системами координат могут оставаться неизменными на протяжении всего развития событий в моделируемой сцене либо меняться с течением времени. Функции связности движений базисных и подчиненных систем координат могут быть линейными и нелинейными.

В следующем подходе отношения между системами координат описываются графом без циклов. В терминах предыдущего подхода это означает, что у системы координат уровня $n + 1$ может быть несколько базисных систем более высоких уровней, т. е. перемещение такой системы координат может определяться с разной функциональной зависимостью перемещениями систем координат более высоких уровней.

И наконец, при моделировании динамических систем с обратной связью отношения между системами координат могут описываться циклическими графами.

Каждый из последних трех подходов является обобщением предыдущего, и поэтому соответствующие им системы координат могут быть легко представлены в терминах расширения типов. Ввиду сравнительной простоты расширения такое представление не приводится в данной статье.

Первый подход, в котором системы координат образуют двухуровневую иерархию, реализуется модулем TWO_LEVEL.

```
MODULE TWO_LEVEL;
TYPE
    SpacePointer* = POINTER TO Space;
    Space* = RECORD location_in_basis- : XYZ.Matrix; END;
(*Поле location_in_basis задает положение системы координат в базисной
системе координат. В двухуровневом подходе базисной для любой системы
координат является абсолютная система координат*)

VAR absolute_space- : SpacePointer;

PROCEDURE (VAR space : Space) LocateInBasis*
                                    (location_in_basis : XYZ.Matrix);
(*Размещение системы координат space в базисной системе координат*)
    BEGIN space.location_in_basis: = location_in_basis; END LocateInBasis;

PROCEDURE (VAR space : Space) RelocateInBasis*
                                    (relocation : XYZ.Matrix);
(*Перемещение системы координат space относительно базисной системы
координат*)
    BEGIN
        XYZ.Compose(space.location_in_basis,
                    space.location_in_basis,
                    relocation);
    END RelocateInBasis;

PROCEDURE (VAR space : Space) RelocateInItself*
                                    (relocation : XYZ.Matrix);
(*Перемещение системы координат space относительно самой себя*)
    BEGIN
        XYZ.Compose(space.location_in_basis,
                    relocation,
                    space.location_in_basis);
    END RelocateInItself;

END TWO LEVEL.
```

Источники освещения. Если для представления поверхностей моделируемой сцены используются грани с интерполяцией освещенности по методу Гуро, то в сцене необходимо разместить источники освещения, для чего естественно использовать те же системы координат. В зависимости от сложности класса моделируемых сцен источники освещения могут быть точечными бесконечно удаленными, точечными на конечном расстоянии, ориентированными, цветными и т. п. Такое усложнение свойств моделей источников освещения также легко выражается в терминах расширения типов данных.

Одна из самых простых моделей, необходимая для последующего изложения, — бесконечно удаленные точечные источники освещения — реализуется модулем INFINITE.

```
MODULE INFINITE;
IMPORT T:= TWO_LEVEL, XYZ;
TYPE
    Lamp* = RECORD
        space- : T.SpacePointer;
        direction- : XYZ.Vector;
    END;
(*Поле space содержит указатель на систему координат источника освещения, поле direction — направление на бесконечно удаленный источник освещения из любой точки в его системе координат*)
PROCEDURE (VAR lamp : Lamp) New*
                                (lamp_space : T.SpacePointer;
                                lamp_direction : XYZ.Vector);
(*Конструктор источника освещения*)
    END New;

END INFINITE.
```

Наблюдатели сцены. Одно из простых представлений наблюдателя задает систему координат, в которой находится наблюдатель, и пирамиду видимости наблюдателя. Это представление реализуется модулем YVIEW.

```
MODULE YVIEW;
IMPORT T:= TWO_LEVEL, WIRE_FACES, XYZ;
TYPE
    Viewer* = POINTER TO ViewerData;
    ViewerData* = RECORD
        space- : T.SpacePointer;
        view_width-,view_height- : REAL;
        pyramid_top-,
        pyramid_left-,
        pyramid_bottom-,
        pyramid_right- : XYZ.Plane;
    END;
```

(*Поле space содержит указатель на систему координат наблюдателя. Считается, что наблюдатель находится в начале системы координат, смотрит вдоль направления оси OY, ось OZ расположена вертикально и направлена вверх, а ось OX — горизонтально и направлена вправо относительно наблюдателя. Линия взгляда наблюдателя перпендикулярна окну наблюдения и проходит через его центр симметрии. Расстояние от начала системы координат наблюдателя до плоскости окна наблюдения не изменяется и равно 1 м. Стороны окна параллельны осям OX и OZ. В полях view_width, view_height задаются соответственно ширина и высота окна наблюдения. Таким образом, расстояние от начала системы координат наблюдателя до окна, размеры и положение окна определяют пирамиду видимости наблюдателя, которой соответствуют значения плоскостей в полях pyramid_top, pyramid_left,

```
pyramid_bottom, pyramid_right. В окне отображается та часть моделируемой
сцены, которая попадает в пирамиду видимости наблюдателя*)

    PROCEDURE (VAR viewer : Viewer) Fill*
                                        (space : T.SpacePointer;
                                        view_width,view_height : REAL);
    (*Конструктор наблюдателя*)
        END New;

    PROCEDURE (v : Viewer) StartView*;
    (*Инициализация вида сцены (начало кадра)*)
        END StartView;

    PROCEDURE (v : Viewer) View* (ff : WIRE_FACES.Faces);
    (*Предъявление наблюдателю граней поверхности или объекта, нахо-
дящихся в моделируемой сцене*)
        END View;

    PROCEDURE (v : Viewer) FinishView*;
    (*Завершение вида сцены (конец кадра)*)
        END FinishView;

    END YVIEW.
```

Статическая модель сцены. Статическая визуальная модель сцены представляется одним изображением или кадром сцены в отличие от динамической визуальной модели, которая представляется последовательностью кадров. Чтобы описать статическую визуальную модель сцены, нужно указать наблюдателя, для которого будет построено изображение сцены, задать условия освещенности сцены, разместить тела и поверхности, соответствующие моделируемым объектам. Процедуры, посредством которых могут быть выполнены перечисленные действия, предоставляются модулем SCENE_FRAME. Этот модуль подразумевает модель вычислений, в которой одному кадру с изображением сцены соответствует исполнение последовательности операторов.

```
    MODULE SCENE_FRAME;
    IMPORT
                F: = FLAT_FACES,
                G: = GOURAUD_FACES,
                I: = INFINITE,
                L: = LIFO,
                S: = SINGLE_VIEW,
                T: = TWO_LEVEL,
                W: = WIRE_FACES;

    VAR
        used_viewer : S.Viewer;
        used_lamp : I.Lamp;
        used_space : T.Space;
    (*used_viewer соответствует текущему наблюдателю, для которого стро-
ится изображение моделируемой сцены в очередном кадре. Начальное зна-
чение used_viewer равно NIL. Наблюдатель задается на момент начала кадра
и не может быть сменен до конца кадра. Началу и концу кадра соответствуют
вызовы процедур StartFrame, FinishFrame (см. далее текст модуля).
        used_lamp задает условия освещенности части (всей) моделируемой сце-
ны. На начало кадра задан источник освещения по умолчанию, находящийся
в начале абсолютной системы координат. Направление на этот источник сов-
падает с положительным направлением оси OY. На протяжении построения
кадра условия освещения частей сцены могут изменяться посредством вызова
процедуры UseNewLamp (см. далее текст модуля)*).
```

used_space соответствует текущей системе координат для предъявления наблюдателю находящихся в моделируемой сцене тел и поверхностей посредством вызова процедуры Show (см. далее текст модуля). На начало кадра в качестве текущей системы координат по умолчанию устанавливается абсолютная система координат*)

```
    PROCEDURE StartFrame* (viewer_for_frame : S.Viewer);
        END StartFrame;

    PROCEDURE FinishFrame*;
        END FinishFrame;

    PROCEDURE UseNewLamp* (New_lamp : I.Lamp);
        END UseNewLamp;

    PROCEDURE UseNewSpace* (New_space : T.Space);
        END UseNewSpace;

    PROCEDURE Show* (ff : W.Faces);
    (*Предъявление набора граней текущему наблюдателю. Грани переводятся из текущей системы координат used_space в систему координат наблюдателя used_viewer.space. Если грань попадает в пирамиду видимости наблюдателя и повернута видимой стороной к нему, то ее изображение будет построено в окне наблюдения на экране дисплея. Для граней типа FLAT_FACES.Face, GOURAUD_FACES.Face интенсивность цвета (освещенность) их поверхности будет рассчитана в соответствии с положением грани относительно источника освещения used_lamp*)
        END Show;

    END SCENE_FRAME.
```

Пример программной модели 3-мерной сцены. В качестве иллюстрации использования определенных выше типов можно рассмотреть программную модель сцены, в которой находится один предмет, представляемый выпуклой оболочкой точек origin, ortX, ortY, ortZ. Этот предмет вращается с частотой один оборот за 360 кадров вокруг оси OZ в собственной системе координат. В модели сцены есть один наблюдатель.

```
    MODULE SIMPLE_SAMPLE;
    IMPORT XYZ, F:=SCENE_FRAME, YVIEW, T:= TWO_LEVEL,
    W:= WIRE_FACES;
    CONST angle = 3.1415926/180;
    VAR
        t,rotate : XYZ.Matrix;
        viewer_shift : XYZ.Vector;
        faces_space,viewer_space : T.SpacePointer;
        viewer : S.Viewer;
        vv : W.VertexList;
        faces : W.Faces;
    BEGIN
    (*Задание системы координат для граней*)
        XYZ.NewMatrix(t,
                      XYZ.ortX,
                      XYZ.ortY,
                      XYZ.ortZ,
                      XYZ.origin);
    NEW(faces_space);
    faces_space.LocateInBasis(t);

    (*Задание системы координат для наблюдателя*)
```

```
    XYZ.NewVector(viewer_shift, 0, -10, 0);
    XYZ.NewMatrix(t,
                  XYZ.ortX,
                  XYZ.ortY,
                  XYZ.ortZ,
                  viewer_shift);
    NEW(viewer_space);
    viewer_space.LocateInBasis(t);
(*Задание наблюдателя с шириной окна наблюдения 1 и высотой 0,8*)
    NEW(viewer);
    viewer.Fill(viewer_space, 1, 0.8);

(*Вычисление матрицы поворота на 1° вокруг оси OZ*)
    XYZ.XtoY(rotate, angle);

(*Создание списка вершин для вращающегося предмета*)
    vv.Default();
    vv.Top(NewVertex(XYZ.ortX));
    vv.Top(NewVertex(XYZ.ortY));
    vv.Top(NewVertex(XYZ.ortZ));
    vv.Top(NewVertex(XYZ.origin));
(*Конструирование выпуклой оболочки вершин из списка vv*)
    faces.Convex(vv);

LOOP (*Генерация бесконечного фильма*)
    F.StartFrame(viewer); (*Начало кадра и задание текущего
                            наблюдателя*)

    F.UseNewSpace(faces_space); (*Установка новой текущей системы
                                  координат объекта*)

    F.Show(faces); (*Предъявление граней наблюдателю*)
    F.FinishFrame; (*Конец кадра*)
    facesSpace.RelocateInBasis(rotate); (*Вращение системы координат
                                          объекта*)
    END;
END SIMPLE_SAMPLE;
```

Заключение. Использование объектно-ориентированного подхода позволяет значительно уменьшить затраты на проектирование, создание, сопровождение и развитие программных моделей динамики трехмерных сцен. Может быть построена иерархия моделей, в которой более сложные модели используют частично или полностью программный код для простых моделей. Основой для данной статьи послужили результаты работ, выполненных к настоящему времени в лаборатории программных систем машинной графики ИАиЭ СО РАН.

СПИСОК ЛИТЕРАТУРЫ

1. **Fiume E.** Object-oriented computer graphics // Advances in Computer Graphics IV /Ed. W. T. Hewitt, M. Grave, M. Roch.—N. Y.: Springer-Verlag, 1991.
2. **Mossenbock H.** Object-oriented Programming in Oberon-2.—N. Y.: Springer-Verlag, 1993.
3. **Magnenat-Thalmann N., Thalmann D.** Image Synthesis: Theory and Practice.—Tokio: Springer-Verlag, 1987.
4. **Gouraud H.** Continuous shading of curved surfaces // Trans. IEEE on Computers.—1971.—20, N 6.—P. 623.

Поступила в редакцию 19 февраля 1993 г.