Национальный исследовательский университет – Высшая школа экономики
Факультет бизнес-информатики, отделение программной инженерии

УТВЕРЖДЕНО
Заведующий кафедрой «Управление разработкой программного обеспечения»
_____ / Авдошин С.М./
«____»_____ 2011 г.

ПРОГРАММА ВИЗУАЛИЗАЦИИ ОПЕРАЦИЙ НАД КВАТЕРНИОНАМИ НА ПЛАТФОРМЕ WINDOWS PHONE 7

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл.	

Руководитель работы
_____ / Гринкруг Е.М./
«____»_____ 2011 г.

Исполнитель: студент группы 171ПИ
_____ / Дубов М.С. /
«____»_____ 2011 г.

2011

Национальный исследовательский университет – Высшая школа экономики
Факультет бизнес-информатики, отделение программной инженерии

УТВЕРЖДЕНО

ПРОГРАММА ВИЗУАЛИЗАЦИИ ОПЕРАЦИЙ НАД КВАТЕРНИОНАМИ НА ПЛАТФОРМЕ WINDOWS PHONE 7

Текст программы

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подп.	

Листов 67

2011

Содержание

1. Библиотека DCL.Maths

1.1. Модуль Quaternion.cs (Работа с кватернионами)

```
using System;
using Microsoft.Xna.Framework;

namespace DCL.Maths
{
    public struct Quaternion
    {
        #region Fields
        //Accessing fields is much faster than properties (!)

        /// <summary>
        /// The scalar (real) part of the quaternion.
        /// </summary>
        public float Re;

        /// <summary>
        /// The i factor of the quaternion.
        /// </summary>
        public float X;

        /// <summary>
        /// The j factor of the quaternion.
        /// </summary>
        public float Y;

        /// <summary>
        /// The k factor of the quaternion.
        /// </summary>
        public float Z;
        #endregion

        #region Constants
            //Quaternions that represent the X, Y and Z axes
            public static readonly Quaternion i = new Quaternion(0, 1, 0, 0);
            public static readonly Quaternion j = new Quaternion(0, 0, 1, 0);
            public static readonly Quaternion k = new Quaternion(0, 0, 0, 1);
        #endregion

        #region Properties
        /// <summary>
        /// Gets the vector part of the quaternion as an instance of the Vector3 Structure (used in WP7)
        /// </summary>
        public Vector3 VectorPart
        {
            get { return new Vector3(X, Y, Z); }
        }

        /// <summary>
        /// The magnitude (absolute value) of the quaternion
        /// </summary>
        public float Abs
        {
            get { return (float)Math.Sqrt(Re * Re + X * X + Y * Y + Z * Z); }
        }
        #endregion

        #region Constructors
        /// <summary>
        /// Creates a quaternion, which has only the scalar (real) part
        /// </summary>
        /// <param name="RealNumber">A real number of single precision</param>
        public Quaternion(float RealNumber)
        {
            Re = RealNumber;
            X = 0;
            Y = 0;
            Z = 0;
        }
        /// <summary>
```

```
/// Creates a quaternion, which has only the vector part
/// </summary>
/// <param name="v">An instance of the Vector3 (used in WP7) structure</param>
public Quaternion(Vector3 v)
{
    Re = 0;
    X = v.X;
    Y = v.Y;
    Z = v.Z;
}
/// <summary>
/// Creates a quaternion by setting all the four factors
/// </summary>
/// <param name="Re">The scalar (real) part of the quaternion</param>
/// <param name="X">The i factor</param>
/// <param name="Y">The i factor</param>
/// <param name="Z">The i factor</param>
public Quaternion(float Re, float X, float Y, float Z)
{
    this.Re = Re;
    this.X = X;
    this.Y = Y;
    this.Z = Z;
}
#endregion

#region Methods
/// <summary>
/// Returns the conjugate of the quaternion
/// </summary>
public Quaternion Conjugate()
{
    return new Quaternion(Re, -X, -Y, -Z);
}
/// <summary>
/// Returns the inverse of the quaternion
/// </summary>
public Quaternion Reciprocal()
{
    if (Re == 0 && X == 0 && Y == 0 && Z == 0)
        throw new DivideByZeroException("Trying to find the reciprocal of a zero number");

    Quaternion temp = this.Conjugate();
    temp.Re /= (Re * Re + X * X + Y * Y + Z * Z);
    temp.X /= (Re * Re + X * X + Y * Y + Z * Z);
    temp.Y /= (Re * Re + X * X + Y * Y + Z * Z);
    temp.Z /= (Re * Re + X * X + Y * Y + Z * Z);

    return temp;
}
/// <summary>
/// Returns the normalized (with the magnitude of 1) quaternion
/// </summary>
public Quaternion Normalize()
{
    float a = Abs;
    return new Quaternion(Re / a, X / a, Y / a, Z / a);
}
/// <summary>
/// Rounds the quaternion, so that each component has the specified amount of signs after dot
/// </summary>
/// <param name="precision">The amount of signs after dot</param>
public Quaternion Round(int precision)
{
    return new Quaternion((float)Common.Round(Re, precision), (float)Common.Round(X, precision),
                          (float)Common.Round(Y, precision), (float)Common.Round(Z, precision));
}
/// <summary>
/// Approximates each component of the quaternion to the specified precision
/// </summary>
/// <param name="precision"></param>
/// <returns></returns>
public Quaternion Approximate(float precision)
{
    float k = 1 / precision;
    return (this * k).Round(0) / k;
```

```
        }
        /// <summary>
        /// Calculates the rotation quaternion from the axis and the rotation angle
        /// </summary>
        /// <param name="Axis">The quaternion that represents axis; it may be denormalized</param>
        /// <param name="Angle">The angle of rotation in radians</param>
        public static Quaternion RotationQuaternion(Quaternion Axis, float Angle)
        {
            Quaternion normAxis = Axis.Normalize();
            float sin = (float)Math.Sin(Angle/2);
            return new Quaternion((float)Math.Cos(Angle / 2),
                                  sin * normAxis.X,
                                  sin * normAxis.Y,
                                  sin * normAxis.Z);
        }
        /// <summary>
        /// Rotates the vector around the axis by the specified angle
        /// </summary>
        /// <param name="Source">The quaternion that represents the vector to be rotated</param>
        /// <param name="Axis">The quaternion that represents axis; it may be denormalized</param>
        /// <param name="Angle">The angle of rotation in radians</param>
        public static Quaternion Rotate(Quaternion Source, Quaternion Axis, float Angle)
        {
            Quaternion qRot = Quaternion.RotationQuaternion(Axis, Angle);
            return (qRot * Source * qRot.Conjugate());
        }
        /// <summary>
        /// Rotates the vector around the translated axis by the specified angle.
        /// </summary>
        /// <param name="Source">The quaternion that represents the vector to be rotated</param>
        /// <param name="Axis">The quaternion that represents axis; it may be denormalized</param>
        /// <param name="Angle">The angle of rotation in radians</param>
        /// <param name="AxisTranslation">The translation of the axis</param>
        public static Quaternion Rotate(Quaternion Source, Quaternion Axis, Quaternion AxisTranslation, float
Angle)
        {
            Quaternion qTemp = Rotate(Source - AxisTranslation, Axis, Angle);
            qTemp += AxisTranslation;
            return qTemp;
        }

        /// <summary>
        /// Makes two rotations of a vector by once.
        /// </summary>
        /// <param name="Source">The quaternion that represents the vector to be rotated</param>
        /// <param name="Axis1">The first axis</param>
        /// <param name="Angle1">The angle of the first rotation in radians</param>
        /// <param name="Axis2">The second axis</param>
        /// <param name="Angle2">The angle of the second rotation in radians</param>
        public static Quaternion RotateComposition(Quaternion Source, Quaternion Axis1, float Angle1,
Quaternion Axis2, float Angle2)
        {
            Quaternion q1 = RotationQuaternion(Axis1, Angle1);
            Quaternion q2 = RotationQuaternion(Axis2, Angle2);
            Quaternion qComb = q2 * q1;

            return qComb * Source  * qComb.Conjugate();

        }

        public static Quaternion Slerp(Quaternion qStart, Quaternion qEnd, float t)
        {
            float CosOm = qStart.X * qEnd.X + qStart.Y * qEnd.Y + qStart.Z * qEnd.Z + qStart.Re * qEnd.Re;
            float Om = (float)Math.Acos(CosOm);

            float Rsqrt = 1 - CosOm * CosOm;

            union u = new union();
            float xhalf = 0.5f * Rsqrt;
            u.asFloat = Rsqrt;
            u.asInt = 0x5f3759df - (u.asInt >> 1);
            Rsqrt = u.asFloat * (1.5f - xhalf * u.asFloat * u.asFloat);

            return ((float)Math.Sin((1 - t) * Om) * qStart + (float)Math.Sin(t * Om) * qEnd) * Rsqrt;
        }
```

```
public static Quaternion LSlerp(Quaternion qStart, Quaternion qEnd, float t)
{
    float CosA = qStart.X * qEnd.X + qStart.Y * qEnd.Y + qStart.Z * qEnd.Z + qStart.Re * qEnd.Re;
    t *= (0.5069269f - CosA * (0.7987229f + 0.5069269f * CosA)) * (t * (2 * t - 3) + 1) + 1;

    Quaternion qRes = qStart + t * (qEnd - qStart);

    float Rsqrt = qRes.X * qRes.X + qRes.Y * qRes.Y + qRes.Z * qRes.Z + qRes.Re * qRes.Re;

    union u = new union();
    float xhalf = 0.5f * Rsqrt;
    u.asFloat = Rsqrt;
    u.asInt = 0x5f3759df - (u.asInt >> 1);
    Rsqrt = u.asFloat * (1.5f - xhalf * u.asFloat * u.asFloat);

    return qRes * Rsqrt;
}

public static Quaternion Lerp(Quaternion qStart, Quaternion qEnd, float t)
{
    Quaternion qRes = qStart + t * (qEnd - qStart);

    float Rsqrt = qRes.X * qRes.X + qRes.Y * qRes.Y + qRes.Z * qRes.Z + qRes.Re * qRes.Re;

    union u = new union();
    float xhalf = 0.5f * Rsqrt;
    u.asFloat = Rsqrt;
    u.asInt = 0x5f3759df - (u.asInt >> 1);
    Rsqrt = u.asFloat * (1.5f - xhalf * u.asFloat * u.asFloat);

    return qRes * Rsqrt;
}

[System.Runtime.InteropServices.StructLayout(System.Runtime.InteropServices.LayoutKind.Explicit)]
private struct union
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int asInt; //32bit

    [System.Runtime.InteropServices.FieldOffset(0)]
    public float asFloat; //32bit
}

#endregion

#region Operators
public static Quaternion operator +(Quaternion l, Quaternion r)
{
    return new Quaternion(l.Re + r.Re, l.X + r.X, l.Y + r.Y, l.Z + r.Z);
}
public static Quaternion operator +(Quaternion l, Vector3 r)
{
    return new Quaternion(l.Re, l.X + r.X, l.Y + r.Y, l.Z + r.Z);
}
public static Quaternion operator +(Vector3 l, Quaternion r)
{
    return new Quaternion(r.Re, l.X + r.X, l.Y + r.Y, l.Z + r.Z);
}
public static Quaternion operator +(Quaternion l, float r)
{
    l.Re += r;
    return l;
}
public static Quaternion operator +(float l, Quaternion r)
{
    r.Re += l;
    return r;
}

public static Quaternion operator -(Quaternion l, Quaternion r)
{
    return new Quaternion(l.Re - r.Re, l.X - r.X, l.Y - r.Y, l.Z - r.Z);
}
public static Quaternion operator -(Quaternion l, Vector3 r)
{
```

```
    return new Quaternion(l.Re, l.X - r.X, l.Y - r.Y, l.Z - r.Z);
}
public static Quaternion operator -(Quaternion l, float r)
{
    l.Re -= r;
    return l;
}

public static Quaternion operator +(Quaternion q)
{
    return q;
}
public static Quaternion operator -(Quaternion q)
{
    return new Quaternion(-q.Re, -q.X, -q.Y, -q.Z);
}

public static Quaternion operator *(Quaternion l, Quaternion r)
{
    return new Quaternion(l.Re * r.Re - l.X * r.X - l.Y * r.Y - l.Z * r.Z,
                          l.Re * r.X + l.X * r.Re + l.Y * r.Z - l.Z * r.Y,
                          l.Re * r.Y + l.Y * r.Re + l.Z * r.X - l.X * r.Z,
                          l.Re * r.Z + l.Z * r.Re + l.X * r.Y - l.Y * r.X);
}
public static Quaternion operator *(Quaternion l, Vector3 r)
{
    return new Quaternion(-l.X * r.X - l.Y * r.Y - l.Z * r.Z,
                          l.Re * r.X + l.Y * r.Z - l.Z * r.Y,
                          l.Re * r.Y + l.Z * r.X - l.X * r.Z,
                          l.Re * r.Z + l.X * r.Y - l.Y * r.X);
}
public static Quaternion operator *(Vector3 l, Quaternion r)
{
    return new Quaternion(-l.X * r.X - l.Y * r.Y - l.Z * r.Z,
                          l.X * r.Re + l.Y * r.Z - l.Z * r.Y,
                          l.Y * r.Re + l.Z * r.X - l.X * r.Z,
                          l.Z * r.Re + l.X * r.Y - l.Y * r.X);
}
public static Quaternion operator *(float l, Quaternion r)
{
    return new Quaternion(l * r.Re, l * r.X, l * r.Y, l * r.Z);
}
public static Quaternion operator *(Quaternion l, float r)
{
    return new Quaternion(l.Re * r, l.X * r, l.Y * r, l.Z * r);
}
public static Quaternion operator /(Quaternion l, float r)
{
    return new Quaternion(l.Re / r, l.X / r, l.Y / r, l.Z / r);
}
public static bool operator ==(Quaternion l, Quaternion r)
{
    return (l.Re == r.Re && l.X == r.X && l.Y == r.Y && l.Z == r.Z);
}
public static bool operator !=(Quaternion l, Quaternion r)
{
    return !(l == r);
}
#endregion

#region Type casting
//The quaternion can be created implicity from a vector or from a real number
public static implicit operator Quaternion(float d)
{
    return new Quaternion(d);
}
public static implicit operator Quaternion(Vector3 v)
{
    return new Quaternion(0, v.X, v.Y, v.Z);
}
//An explicit convert from the quaternion to a vector; the real-part may be non-zero
public static explicit operator Vector3(Quaternion q)
{
    return new Vector3(q.X, q.Y, q.Z);
}
#endregion
```

```
        #region Overridden methods
        /// <summary>
        /// Returns the string representation of the quaternion
        /// </summary>
        public override string ToString()
        {
            string fmt = "";
            if (Re > 0)
                fmt += Re.ToString()+" ";
            else if (Re < 0)
                fmt += ("- " + (-Re).ToString()) + " ";

            if (X > 0)
                fmt += String.Format("{0}{1}i ", (fmt=="") ? "" : "+ ", (X == 1) ? "" : X.ToString());
            else if (X < 0)
                fmt += String.Format("- {0}i ", (X == -1) ? "" : (-X).ToString());

            if (Y > 0)
                fmt += String.Format("{0}{1}j ", (fmt == "") ? "" : "+ ", (Y == 1) ? "" : Y.ToString());
            else if (Y < 0)
                fmt += String.Format("- {0}j ", (Y == -1) ? "" : (-Y).ToString());

            if (Z > 0)
                fmt += String.Format("{0}{1}k", (fmt == "") ? "" : "+ ", (Z == 1) ? "" : Z.ToString());
            else if (Z < 0)
                fmt += String.Format("- {0}k", (Z == -1) ? "" : (-Z).ToString());

            if (fmt == "") fmt = "0";

            return fmt;
        }

        /// <summary>
        /// Returns the string representation of the quaternion with specified precision
        /// </summary>
        public string ToString(int Precision)
        {
            string fmt = "";
            if (Re > 0)
                fmt += Common.Round(Re, Precision).ToString() + " ";
            else if (Re < 0)
                fmt += ("- " + (-Common.Round(Re, Precision)).ToString()) + " ";

            if (X > 0)
                fmt += String.Format("{0}{1}i ", (fmt == "") ? "" : "+ ", (X == 1) ? "" : Common.Round(X,
Precision).ToString());
            else if (X < 0)
                fmt += String.Format("- {0}i ", (X == -1) ? "" : (-Common.Round(X, Precision)).ToString());

            if (Y > 0)
                fmt += String.Format("{0}{1}j ", (fmt == "") ? "" : "+ ", (Y == 1) ? "" : Common.Round(Y,
Precision).ToString());
            else if (Y < 0)
                fmt += String.Format("- {0}j ", (Y == -1) ? "" : (-Common.Round(Y, Precision)).ToString());

            if (Z > 0)
                fmt += String.Format("{0}{1}k", (fmt == "") ? "" : "+ ", (Z == 1) ? "" : Common.Round(Z,
Precision).ToString());
            else if (Z < 0)
                fmt += String.Format("- {0}k", (Z == -1) ? "" : (-Common.Round(Z, Precision)).ToString());

            if (fmt == "") fmt = "0";

            return fmt;
        }

        /// <summary>
        /// Defines if the current instance of quaternion is equal to the parameter
        /// </summary>
        public override bool Equals(object obj)
        {
            return obj is Quaternion && this == (Quaternion)obj;
        }

        /// <summary>
```

```
        /// Returns the hash code of the current instance of quaternion
        /// </summary>
        public override int GetHashCode()
        {
            return (int)Re ^ (int)X ^ (int)Y ^ (int)Z;
        }
        #endregion
    }
}
```

1.2. Модуль Fraction.cs (Работа с обыкновенными дробями)

```
using System;

namespace DCL.Maths
{
    public struct Fraction: IComparable
    {
        #region Fields
        long num;
        uint den;
        #endregion

        #region Constants
        public static readonly Fraction Empty = new Fraction(long.MaxValue, uint.MaxValue);
        #endregion

        #region Properties
        public long Numerator
        {
            set { num = value; }
            get { return num; }
        }
        public uint Denominator
        {
            set
            {
                if (value == 0) throw new DivideByZeroException("denominator");
                den = value;
            }
            get { return den; }
        }
        #endregion

        #region Constructors
        public Fraction(long numerator, uint denominator)
        {
            num = numerator;
            if (denominator == 0) throw new DivideByZeroException("denominator");
            den = denominator;
        }
        #endregion

        #region Operators
        public static Fraction operator +(Fraction l, Fraction r)
        {
            if (l.Numerator == Empty.Numerator || l.Denominator == Empty.Denominator)
                if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
                {
                    Fraction f = 0;
                    return Empty;
                }
                else
                    return r;
            if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
                return l;

            uint CommonDenominator = Common.LCM(l.Denominator, r.Denominator);
            Fraction fr = new Fraction(l.Numerator * CommonDenominator / l.Denominator
                                       + r.Numerator * CommonDenominator / r.Denominator,
                                       CommonDenominator);
            return fr.Cancel();
```

```
}
public static Fraction operator -(Fraction l, Fraction r)
{
    if (l.Numerator == Empty.Numerator || l.Denominator == Empty.Denominator)
        if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
            return Empty;
        else
            return r;
    if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
        return l;

    uint CommonDenominator = Common.LCM(l.Denominator, r.Denominator);
    Fraction fr = new Fraction(l.Numerator * CommonDenominator / l.Denominator
                               - r.Numerator * CommonDenominator / r.Denominator,
                               CommonDenominator);
    return fr.Cancel();
}
public static Fraction operator *(Fraction l, Fraction r)
{
    if (l.Numerator == Empty.Numerator || l.Denominator == Empty.Denominator)
        if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
            return Empty;
        else
            return r;
    if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
        return l;

    Fraction fr = new Fraction(l.Numerator * r.Numerator,
                               l.Denominator * r.Denominator);
    return fr.Cancel();
}
public static Fraction operator /(Fraction l, Fraction r)
{
    return l*r.Reverse();
}
public static Fraction operator +(Fraction f)
{
    return new Fraction(f.Numerator, f.Denominator);
}
public static Fraction operator -(Fraction f)
{
    if (f.Numerator == Empty.Numerator || f.Denominator == Empty.Denominator)
        return f;

    return new Fraction(-f.Numerator, f.Denominator);
}
public static Fraction operator ++(Fraction f)
{
    if (f.Numerator == Empty.Numerator || f.Denominator == Empty.Denominator)
        return f;

    return new Fraction(f.Numerator + f.Denominator, f.Denominator);
}
public static Fraction operator --(Fraction f)
{
    if (f.Numerator == Empty.Numerator || f.Denominator == Empty.Denominator)
        return f;

    return new Fraction(f.Numerator - f.Denominator, f.Denominator);
}
public static bool operator ==(Fraction l, Fraction r)
{
    return (l.Cancel().Numerator == r.Cancel().Numerator &&
            l.Cancel().Denominator == r.Cancel().Denominator) ||
            ((l.Numerator==Empty.Numerator || l.Denominator==Empty.Denominator) &&
             (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator));
}
public static bool operator >(Fraction l, Fraction r)
{
    if (l.Numerator == Empty.Numerator || l.Denominator == Empty.Denominator)
        return false;
    else if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
        return true;

    if (r.Numerator == 0) return l.Numerator > 0;
    if (l.Numerator == 0) return r.Numerator < 0;
```

```
            uint CommonDenominator = Common.LCM(l.Denominator, r.Denominator);
            return l.Numerator * CommonDenominator / l.Denominator >
                    r.Numerator * CommonDenominator / r.Denominator;
        }
        public static bool operator >=(Fraction l, Fraction r)
        {
            if (l.Numerator == Empty.Numerator || l.Denominator == Empty.Denominator)
                if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
                    return true;
                else
                    return false;
            if (r.Numerator == Empty.Numerator || r.Denominator == Empty.Denominator)
                return true;

            if (r.Numerator == 0) return l.Numerator >= 0;
            if (l.Numerator == 0) return r.Numerator <= 0;

            uint CommonDenominator = Common.LCM(l.Denominator, r.Denominator);
            return l.Numerator * CommonDenominator / l.Denominator >=
                    r.Numerator * CommonDenominator / r.Denominator;
        }
        public static bool operator <(Fraction l, Fraction r)
        {
            return r > l;
        }
        public static bool operator <=(Fraction l, Fraction r)
        {
            return r >= l;
        }
        public static bool operator !=(Fraction l, Fraction r)
        {
            return !(l == r);
        }
        #endregion

        #region Type casting
        public static implicit operator Fraction(long l)
        {
            return new Fraction(l,1);
        }
        public static explicit operator Fraction(double d)
        {
            if (d == 0) return new Fraction(0, 1);

            int l = Math.Min(Common.FractionalPartLength(d), 8);//int l = Math.Min(19 -
(int)Math.Log10(Math.Abs(d)), 8);
            uint dn = (uint)Math.Pow(10, l);
            long nm = (long)(d * dn);
            Fraction fr = new Fraction(nm, dn);

            return fr;//fr.Cancel();
        }
        public static explicit operator double(Fraction f)
        {
            if (f.Numerator == Fraction.Empty.Numerator || f.Denominator == Fraction.Empty.Denominator)
                return 0;

            return (double)f.Numerator / f.Denominator;
        }
        #endregion

        #region Methods
        public Fraction Cancel()
        {
            uint t = Common.GCD((uint)Math.Abs(num), den);
            return new Fraction(num/t, den/t);
        }
        public Fraction Reverse()
        {
            if (num == 0) throw new DivideByZeroException("denominator");

            Fraction f = new Fraction();

            if (den == Empty.Denominator) f.Numerator = Empty.Numerator;
            else f.Numerator = (num < 0) ? -den : den;
```

```
            if (num == Empty.Numerator) f.Denominator = Empty.Denominator;
            else f.Denominator = (uint)Math.Abs(num);

            return f;
        }
        public Fraction ReduceToDenominator(uint newDenominator)
        {
            return new Fraction(Numerator * newDenominator / Denominator, newDenominator);
        }
        public void ToMixedNumber(out long WholePart, out Fraction FractionalPart)
        {
            if (this.Numerator == Empty.Numerator || this.Denominator == Empty.Denominator)
            {
                WholePart = 0;
                FractionalPart = Empty;
                return;
            }
            bool neg = this.Numerator < 0;
            WholePart = this.Numerator / this.Denominator;
            FractionalPart = (new Fraction(Math.Abs(this.Numerator) - Math.Abs(WholePart * this.Denominator),
this.Denominator)).Cancel();
            if (WholePart == 0 && neg)
                FractionalPart.Numerator *= -1;
        }
        public static Fraction FromMixedNumber(long wholePart, Fraction fractionalPart)
        {
            if (fractionalPart.Numerator == Fraction.Empty.Numerator || fractionalPart.Denominator ==
Fraction.Empty.Denominator)
                return Fraction.Empty;

            bool neg = wholePart < 0 || fractionalPart.Numerator < 0;
            if (wholePart == 0 && neg) fractionalPart *= -1;
            if (wholePart < 0) wholePart *= -1;
            Fraction f = wholePart + fractionalPart;
            if (neg) f *= -1;
            return f;
        }
        public static Fraction Parse(string str)
        {
            long nm;
            uint dn;

            str = str.Trim();

            if (str.IndexOf('/') == -1) { nm = long.Parse(str); dn = 1; }
            else
            {
                nm = long.Parse(str.Substring(0, str.IndexOf('/')));
                dn = uint.Parse(str.Substring(str.IndexOf('/') + 1, str.Length - str.IndexOf('/')-1));
            }

            return new Fraction(nm, dn);
        }
        public static bool TryParse(string str, out Fraction f)
        {
            try
            {
                f = Parse(str);
                return true;
            }
            catch
            {
                f = new Fraction(0,1);
                return false;
            }
        }
        public static bool IsEmpty(Fraction f)
        {
            return f.Numerator == Fraction.Empty.Numerator || f.Denominator == Fraction.Empty.Denominator;
        }
        #endregion

        #region Overridden methods
        public override bool Equals(object obj)
        {
```

```
            return (obj is Fraction) && (this == (Fraction)obj);
        }

        public override int GetHashCode()
        {
            return (num/den).GetHashCode();
        }

        public override string ToString()
        {
            return String.Format("{0}/{1}",
                (num == Empty.Numerator) ? "" : num.ToString(),
                (den == Empty.Denominator) ? "" : den.ToString());
        }
        #endregion

        #region IComparable interface realization
        public int CompareTo(object obj)
        {
            if (obj == null) return 1;

            if (!(obj is Fraction)) throw new ArgumentException();

            if (this > (Fraction)obj) return 1;
            else if (this < (Fraction)obj) return -1;
            else return 0;
        }
        #endregion
    }
}
```

1.3. Модуль Angle.cs (Работа с углами)

```
using System;

namespace DCL.Maths
{
    public struct Angle: IComparable
    {
        //Данная реализация структуры градусов
        //представляет собой как бы "обертку"
        //над структурой Fraction
        internal Fraction val;

        #region Constants
        public static readonly Angle PI = new Angle(180,1);
        public static readonly Angle Eps = new Angle(0,0,1, 1);
        #endregion

        #region Properties
        public int Sign
        {
            get
            {
                if (val < 0) return -1;
                if (val > 0) return 1;
                return 0;
            }
        }

        public uint Degrees
        {
            set
            {
                val = value + new Fraction(Minutes, 60) + new Fraction(Seconds, 3600);
            }
            get { return (uint)Math.Abs((double)val.ReduceToDenominator(3600)); }
        }
        public uint Minutes
        {
            set
            {
                if(value>60) throw new ArgumentOutOfRangeException("Minutes");
```

```
            val = Degrees + new Fraction(value, 60) + new Fraction(Seconds, 3600);
        }
        get { return (uint)(Math.Abs((val.ReduceToDenominator(3600).Numerator) % 3600) / 60); }
    }
    public uint Seconds
    {
        set
        {
            if (value > 60) throw new ArgumentOutOfRangeException("Minutes");

            val = Degrees + new Fraction(Minutes, 60) + new Fraction(value, 3600);
        }
        get { return (uint)(Math.Abs(val.ReduceToDenominator(3600).Numerator) % 60); }
    }
    #endregion

    #region Constructors
    public Angle(uint degrees, int sign)
    {
        val = degrees;
        if (sign < 0) val = -val;
    }
    public Angle(uint degrees, uint minutes, uint seconds, int sign)
    {
        val = degrees + new Fraction(minutes, 60) + new Fraction(seconds, 3600);
        if (sign < 0) val = -val;
    }
    private Angle(Fraction f)
    {
        val = f;
    }
    #endregion

    #region Operators
    public static Angle operator +(Angle l, Angle r)
    {
        Fraction res = (l.val + r.val);
        return new Angle(res);
    }
    public static Angle operator -(Angle l, Angle r)
    {
        Fraction res = (l.val - r.val);
        return new Angle(res);
    }
    public static Angle operator *(Angle l, Angle r)
    {
        return (Angle)((double)l * (double)r);
    }
    public static Angle operator *(Angle l, int r)
    {
        Fraction res = l.val * r;
        return new Angle(res);
    }
    public static Angle operator *(int l, Angle r)
    {
        return r * l;
    }
    public static Angle operator /(Angle l, Angle r)
    {
        return (Angle)((double)l / (double)r);
    }
    public static Angle operator /(Angle l, int r)
    {
        Fraction res = l.val / r;
        return new Angle(res);
    }
    public static Angle operator +(Angle a)
    {
        return new Angle(a.val);
    }
    public static Angle operator -(Angle a)
    {
        return new Angle(-a.val);
    }
    public static Angle operator ++(Angle a)
    {
```

```
        return new Angle(a.val + 1);
    }
    public static Angle operator --(Angle a)
    {
        return new Angle(a.val - 1);
    }
    public static bool operator >(Angle l, Angle r)
    {
        return (l.val > r.val);
    }
    public static bool operator >=(Angle l, Angle r)
    {
        return (l.val >= r.val);
    }
    public static bool operator <(Angle l, Angle r)
    {
        return (l.val < r.val);
    }
    public static bool operator <=(Angle l, Angle r)
    {
        return (l.val <= r.val);
    }
    public static bool operator ==(Angle l, Angle r)
    {
        return l.val == r.val;
    }
    public static bool operator !=(Angle l, Angle r)
    {
        return !(l == r);
    }
    #endregion

    #region Type casting
    public static explicit operator double(Angle a)
    {
        return (double)a.val;
    }
    public static explicit operator Angle(double d)
    {
        return new Angle((Fraction)d);
    }
    public static explicit operator Fraction(Angle a)
    {
        return a.val;
    }
    public static explicit operator Angle(Fraction f)
    {
        return new Angle(f);
    }
    #endregion

    #region Methods
    public static Angle Parse(string str)
    {
        str = str.Trim();
        uint deg, min=0, sec=0;
        int sign=1;

        if (str.IndexOf('°') < 0)
            throw new ArgumentException();

        if (str[0] == '-')
        {
            sign = -1;
            str = str.Substring(1);
        }

        deg = UInt32.Parse(str.Substring(0, str.IndexOf('°')));

        if (str.IndexOf('\'') > 0)
        {
            str = str.Substring(str.IndexOf('°') + 1);
            min = UInt32.Parse(str.Substring(0, str.IndexOf('\'')));
        }
        if (str.IndexOf('\"') > 0)
        {
```

```
            str = str.Substring(str.IndexOf('\'') + 1);
            min = UInt32.Parse(str.Substring(0, str.IndexOf('\"')));
        }

        return new Angle(deg, min, sec, sign);
    }
    public static bool TryParse(string str, out Angle a)
    {
        try
        {
            a = Parse(str);
            return true;
        }
        catch
        {
            a = (Angle)0;
            return false;
        }
    }
    public static Angle Wrap(Angle a)
    {
        if(a.Sign>0)
            while (a.Degrees > 180)
                a -= new Angle(360, 1);
        else
            while (a.Degrees > 180)
                a += new Angle(360, 1);

        return a;
    }
    #endregion

    #region Overridden methods
    public override bool Equals(object obj)
    {
        return (obj is Angle) && (this == (Angle)obj);
    }

    public override int GetHashCode()
    {
        return ((double)this).GetHashCode();
    }
    public override string ToString()
    {
        if (Minutes == 0 && Seconds == 0)
            return String.Format("{0}{1}°", (val < 0)?"-":"", Degrees);
        else if(Seconds==0)
            return String.Format("{0}{1}° {2}\'", (val < 0) ? "-" : "", Degrees, Minutes);
        else
            return String.Format("{0}{1}° {2}\' {3}\"", (val < 0) ? "-" : "", Degrees, Minutes, Seconds);
    }
    public string ToString(bool AlwaysPrintMinutes, bool AlwaysPrintSeconds)
    {
        string str = String.Format("{0}{1}°", (val < 0) ? "-" : "", Degrees);

        if(Minutes!=0 || AlwaysPrintMinutes || Seconds!=0)
            str += String.Format(" {0}\'", Minutes);

        if (Seconds != 0 || AlwaysPrintSeconds)
            str += String.Format(" {0}\"", Seconds);

        return str;
    }
    #endregion

    #region IComparable interface realization
    public int CompareTo(object obj)
    {
        if (obj == null) return 1;

        if (!(obj is Angle)) throw new ArgumentException();

        if (this > (Angle)obj) return 1;
        else if (this < (Angle)obj) return -1;
        else return 0;
    }
```

```
        #endregion
    }
}
```

1.4. Модуль Common.cs (Общие математические методы)

```
using System;

namespace DCL.Maths
{
    /// <summary>
    /// A static class that contains several common mathematical methods
    /// and extends the functionality of Math.Round.
    /// </summary>
    public static class Common
    {
        /// <summary>
        /// Rounds a number in mathematical rules.
        /// Unlike Math.Round, Common.Round(34.45, 1) returns 34.5 instead of 34.4.
        /// </summary>
        /// <param name="number">A fractional number to be rounded.</param>
        /// <param name="precision">The number of decimal places in the return value.</param>
        /// <returns>A double value.</returns>
        public static double Round(double number, int precision)
        {
            double temp = Math.Pow(10, precision);
            return Math.Floor(number * temp + 0.5) / temp;
        }

        /// <summary>
        /// Rounds a number in mathematical rules.
        /// Unlike Math.Round, Common.Round(34.5) returns 35 instead of 34.
        /// </summary>
        /// <param name="number">A fractional number to be rounded.</param>
        /// <returns>A double value.</returns>
        public static double Round(double number)
        {
            return Math.Floor(number+0.5);
        }

        /// <summary>
        /// Calculates the cube root of a number.
        /// </summary>
        /// <param name="x">The initial double value.</param>
        /// <returns>A double value.</returns>
        public static double Cbrt(double x)
        {
            double a1, //Previous value
                a2, //Current value
                eps; //Next value

            double standartPrecision = 1e-10;

            a2 = x;
            do
            {
                a1 = a2;
                eps = (x - a1 * a1 * a1) / (3 * a1 * a1);
                a2 = a1 + eps;
            } while (Math.Abs(eps)>standartPrecision); //till reaching the "machine null"
            return a2;
        }

        /// <summary>
        /// Calculates the factorial of a natural number.
        /// </summary>
        /// <param name="x">The initial natural value; the value can be zero.</param>
        /// <returns>A natural value.</returns>
        public static double Factorial(uint x)
        {
            //if (x > 65) throw new OverflowException("Too large!");
            if (x == 0) return 1;
            else
            {
```

```
        double res = 1;
        for (uint i = 2; i <= x; i++)
            res *= i;
        return res;
    }
}

/// <summary>
/// Calculates the double factorial of a natural number.
/// </summary>
/// <param name="x">The initial natural value; the value can be zero.</param>
/// <returns>A natural value.</returns>
public static ulong DoubleFactorial(uint x)
{
    if (x == 0) return 1;
    else
    {
        ulong res = 1;
        for (uint i = (x%2==0)?2u:3u; i <= x; i+=2)
            res *= i;
        return res;
    }
}

/// <summary>
/// Calculates the least common multiple of two natural numbers.
/// </summary>
/// <param name="a">The first natural number.</param>
/// <param name="b">The second natural number.</param>
/// <returns>A natural value.</returns>
public static uint LCM(uint a, uint b) //НОК
{
    return a*b/GCD(a,b);
}

/// <summary>
/// Calculates the greatest common divisor of two natural numbers.
/// </summary>
/// <param name="a">The first natural number.</param>
/// <param name="b">The second natural number.</param>
/// <returns>A natural value.</returns>
public static uint GCD(uint a, uint b) //НОД
{
    if (a == 0 || b == 0) return Math.Max(a, b);
    while (a != b)
        if (a > b) a -= b;
        else b -= a;
    return a;
}
/*
public static uint GCD(uint a, uint b) //Faster when handling large values
{
    if (a == 0 || b == 0) return Math.Max(a, b);
    if (a > b) Swap(ref a, ref b);

    while (b != 0)
    {
        uint r = a % b;
        a = b;
        b = r;
    }
    return a;
}*/


/// <summary>
/// Calculates the fractional part of a number.
/// </summary>
/// <param name="x">The initial fractional number.</param>
/// <returns>A double value, the integral part of which is equal to zero.</returns>
public static double FractionalPart(double x)
{
    return (Math.Abs(x) - Math.Abs((long)x));
}

/// <summary>
```

```
/// Calculates the fractional part of a number.
/// </summary>
/// <param name="x">The initial fractional number.</param>
/// <returns>A decimal value, the integral part of which is equal to zero.</returns>
public static decimal FractionalPart(decimal x)
{
    return (Math.Abs(x) - Math.Abs((long)x));
}

/// <summary>
/// Calculates the length of the fractional part of a number.
/// </summary>
/// <param name="x">The initial fractional number.</param>
/// <returns>An integer value.</returns>
public static int FractionalPartLength(double x)
{
    return FractionalPartLength((decimal)x); //should pe precise!!
    /*int l = 0;
    while (FractionalPart(x) > 0)
    {
        x *= 10;
        l++;
    }
    return l;*/
}

/// <summary>
/// Calculates the length of the fractional part of a number.
/// </summary>
/// <param name="x">The initial fractional number.</param>
/// <returns>An integer value.</returns>
public static int FractionalPartLength(decimal x)
{
    int l = 0;
    while (FractionalPart(x) > 0)
    {
        x *= 10;
        l++;
    }
    return l;
}

/// <summary>
/// Defines if an unsigned integer is a prime number.
/// </summary>
/// <param name="number">A number to be analyzed.</param>
/// <returns>True if prime; false otherwise.</returns>
public static bool IsPrime(uint number)
{
    int temp = (int)Math.Sqrt(number);
    for (int i = 2; i <= temp; i++)
        if (number % i == 0) return false;
    return true;
}

/// <summary>
/// Swaps the values of two variables.
/// </summary>
/// <param name="a">The first variable.</param>
/// <param name="b">The second variable.</param>
public static void Swap(ref uint a, ref uint b)
{
    uint t = a;
    a = b;
    b = t;
}

/// <summary>
/// Swaps the values of two variables.
/// </summary>
/// <param name="a">The first variable.</param>
/// <param name="b">The second variable.</param>
public static void Swap(ref int a, ref int b)
{
    int t = a;
    a = b;
```

```
            b = t;
        }

        /// <summary>
        /// Calculates the binomial coeffitient (which is also the amount of k-combinations of n)
        /// </summary>
        public static double BinomialCoefficient(uint n, uint k)
        {
            uint max = Math.Max(k, n - k);
            uint min = Math.Min(k, n - k);

            double res = 1;

            while (n > max)
                res *= n--;

            if (Double.IsPositiveInfinity(res)) return Double.PositiveInfinity;

            double fact = Factorial(min);
            if (Double.IsPositiveInfinity(fact)) return Double.PositiveInfinity;

            res /= Factorial(min);

            return res;
        }
    }
}
```

2. Библиотека DCL.Phone.Xna

2.1. Классы, отвечающие за построение трехмерных фигур

2.1.1. Модуль Shape.cs (Представление трехмерной фигуры)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;
using My=DCL.Maths;

namespace DCL.Phone.Xna
{
    /// <summary>
    /// The enumeration of ways in which the shape can be drawn.
    /// </summary>
    public enum DrawMode
    {
        /// <summary>
        /// Draws the outline of the shape only.
        /// </summary>
        Lines = 0x001,

        /// <summary>
        /// Draws the shape's surface.
        /// </summary>
        Solid = 0x010,

        /// <summary>
        /// Draws both the surface and the outline of the shape.
        /// </summary>
        SolidWithLines = 0x011,     // = Solid | Lines

        /// <summary>
        /// Draws the texture on the shape's surface.
```

```
        /// </summary>
        Textured = 0x100,
        //TexturedWithLines = 0x101   // = Textured | Lines
    }

    /// <summary>
    /// An abstract class that represents a three-dimensional shape and has methods
    /// to draw and rotate it.
    /// </summary>
    public abstract class Shape
    {
        #region Fields
        /// <summary>
        /// The array that stores the shape's vertices after the transformations occured.
        /// These vertices are used while drawing the shape.
        /// </summary>
        protected VertexPositionNormalTexture[] currentVertices;
        /// <summary>
        /// The array that stores the shape's initial vertices.
        /// Calling Reset() copies the values from this array to currentVertices.
        /// </summary>
        protected VertexPositionNormalTexture[] startVertices;

        /// <summary>
        /// The triangle indices array that is used when drawing in solid and texture modes.
        /// </summary>
        protected short[] triangleIndices;
        /// <summary>
        /// The line indices array that is used when drawing in line mode.
        /// </summary>
        protected short[] lineIndices;

        /// <summary>
        /// The initial center of the shape.
        /// Calling Reset() copies this value to the Center property.
        /// </summary>
        protected Vector3 startCenter;

        private My.Quaternion qDefaultRotation = My.Quaternion.RotationQuaternion(My.Quaternion.j, 0.05f);
        private My.Quaternion qDefaultRotationConj = My.Quaternion.RotationQuaternion(My.Quaternion.j,
0.05f).Conjugate();
        #endregion

        #region Properties
        /// <summary>
        /// Gets or sets the center point of the shape.
        /// </summary>
        public Vector3 Center { get; protected set; }

        /// <summary>
        /// Gets or sets the texture that is drawn to the surface of shape
        /// </summary>
        public Texture2D Texture { get; set; }

        /// <summary>
        /// Gets or sets the graphics device that is responsible for drawing the shape
        /// </summary>
        public GraphicsDevice GraphicsDevice { get; set; }


        /// <summary>
        /// Gets or sets the quaternion that represents the default rotation for the shape.
        /// This quaternion is used in the RotateDefault() method.
        /// </summary>
        protected My.Quaternion DefaultRotation
        {
            get { return qDefaultRotation; }
            set
            {
                qDefaultRotation = value;
                qDefaultRotationConj = value.Conjugate();
            }
        }
        #endregion

        #region Constructors
```

```
        /// <summary>
        /// Constructor that initializes the center with zero vector.
        /// </summary>
        protected Shape()
        {
            Center = startCenter = Vector3.Zero;
        }
        #endregion

        #region Non-static methods
        /// <summary>
        /// Returns the shape to its initial state.
        /// </summary>
        public void Reset()
        {
            Array.Copy(startVertices, currentVertices, currentVertices.Length);
            Center = startCenter;
        }

        /// <summary>
        /// Rotates the shape around an axis over a specified angle. The rotation is based on quaternions.
        /// </summary>
        /// <param name="Axis">A quaternion or a Vector3 object that represents the axis. It is assumed that
the axis starts at the origin.</param>
        /// <param name="RotationAngle">The angle of rotation.</param>
        public void Rotate(My.Quaternion Axis, float RotationAngle)
        {
            if (RotationAngle == 0) return;

            My.Quaternion qRot = My.Quaternion.RotationQuaternion(Axis, RotationAngle);
            My.Quaternion qRotConj = qRot.Conjugate();
            for (int i = 0; i < currentVertices.Length; i++)
            {
                currentVertices[i] = new VertexPositionNormalTexture(
                    (Vector3)(qRot * currentVertices[i].Position * qRotConj),
                        Vector3.Up, currentVertices[i].TextureCoordinate);
            }
            Center = (Vector3)(qRot * Center * qRotConj);
        }

        /// <summary>
        /// Rotates the shape around an axis over a specified angle. The rotation is based on quaternions.
        /// </summary>
        /// <param name="Axis">A quaternion or a Vector3 object that represents the axis.</param>
        /// <param name="Translation">A quaternion or a Vector3 object that represents the point at which the
axis starts.</param>
        /// <param name="RotationAngle">The angle of rotation.</param>
        public void Rotate(My.Quaternion Axis, My.Quaternion Translation, float RotationAngle)
        {
            if (RotationAngle == 0) return;

            My.Quaternion qRot = My.Quaternion.RotationQuaternion(Axis, RotationAngle);
            My.Quaternion qRotConj = qRot.Conjugate();
            for (int i = 0; i < currentVertices.Length; i++)
            {
                currentVertices[i] = new VertexPositionNormalTexture(
                    (Vector3)(qRot * (currentVertices[i].Position - Translation) * qRotConj + Translation),
                        Vector3.Up, currentVertices[i].TextureCoordinate);
            }
            Center = (Vector3)(qRot*(Center - Translation)*qRotConj + Translation);
        }

        /// <summary>
        /// Rotates the shape around two axises over two specified angles. The rotation is based on
quaternions.
        /// </summary>
        /// <param name="Axis1">A quaternion or a Vector3 object that represents the first axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle1">The angle of rotation around the first axis.</param>
        /// <param name="Axis2">A quaternion or a Vector3 object that represents the second axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle2">The angle of rotation around the second axis.</param>
        public void RotateComposition(My.Quaternion Axis1, float RotationAngle1,
                                      My.Quaternion Axis2, float RotationAngle2)
        {
            My.Quaternion qRot1 = My.Quaternion.RotationQuaternion(Axis1, RotationAngle1);
```

```
            My.Quaternion qRot2 = My.Quaternion.RotationQuaternion(Axis2, RotationAngle2);
            My.Quaternion qComb = qRot2 * qRot1;
            My.Quaternion qCombConj = qComb.Conjugate();

            for (int i = 0; i < currentVertices.Length; i++)
            {
                currentVertices[i] = new VertexPositionNormalTexture(
                    (Vector3)(qComb * currentVertices[i].Position * qCombConj),
                        Vector3.Up, currentVertices[i].TextureCoordinate);
            }
            Center = (Vector3)(qComb*Center*qCombConj);
        }

        /// <summary>
        /// Rotates the shape around two axises over two specified angles. The rotation is based on
quaternions.
        /// </summary>
        /// <param name="Axis1">A quaternion or a Vector3 object that represents the first axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle1">The angle of rotation around the first axis.</param>
        /// <param name="Axis2">A quaternion or a Vector3 object that represents the second axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle2">The angle of rotation around the second axis.</param>
        /// <param name="Axis3">A quaternion or a Vector3 object that represents the third axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle3">The angle of rotation around the third axis.</param>
        public void RotateComposition(My.Quaternion Axis1, float RotationAngle1,
                                      My.Quaternion Axis2, float RotationAngle2,
                                      My.Quaternion Axis3, float RotationAngle3)
        {
            My.Quaternion qRot1 = My.Quaternion.RotationQuaternion(Axis1, RotationAngle1);
            My.Quaternion qRot2 = My.Quaternion.RotationQuaternion(Axis2, RotationAngle2);
            My.Quaternion qRot3 = My.Quaternion.RotationQuaternion(Axis3, RotationAngle3);
            My.Quaternion qComb = qRot3 * qRot2 * qRot1; //Quaternion's multiplication is accotiative
            My.Quaternion qCombConj = qComb.Conjugate();

            for (int i = 0; i < currentVertices.Length; i++)
            {
                currentVertices[i] = new VertexPositionNormalTexture(
                    (Vector3)(qComb * currentVertices[i].Position * qCombConj),
                        Vector3.Up, currentVertices[i].TextureCoordinate);
            }
            Center = (Vector3)(qComb * Center * qCombConj);
        }

        /// <summary>
        /// Performs the default rotation, which can be changed through the SetDefaultRotation method.
        /// Is preferable in cases when the axises and angles don't change each frame.
        /// </summary>
        public void RotateDefault()
        {
            for (int i = 0; i < currentVertices.Length; i++)
            {
                currentVertices[i] = new VertexPositionNormalTexture(
                    (Vector3)(qDefaultRotation * currentVertices[i].Position * qDefaultRotationConj),
                        Vector3.Up, currentVertices[i].TextureCoordinate);
            }
            Center = (Vector3)(qDefaultRotation * Center * qDefaultRotationConj);
        }

        /// <summary>
        /// Sets the default rotation for the RotateDefault() method.
        /// </summary>
        /// <param name="Axis">A quaternion or a Vector3 object that represents the axis.</param>
        /// <param name="RotationAngle">The angle of rotation around the axis.</param>
        public void SetDefaultRotation(My.Quaternion Axis, float RotationAngle)
        {
            DefaultRotation = My.Quaternion.RotationQuaternion(Axis, RotationAngle);
        }

        /// <summary>
        /// Sets the default rotation for the RotateDefault() method.
        /// </summary>
        /// <param name="Axis1">A quaternion or a Vector3 object that represents the first axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle1">The angle of rotation around the first axis.</param>
```

```
        /// <param name="Axis2">A quaternion or a Vector3 object that represents the second axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle2">The angle of rotation around the second axis.</param>
        public void SetDefaultRotation(My.Quaternion Axis1, float RotationAngle1,
                                       My.Quaternion Axis2, float RotationAngle2)
        {
            My.Quaternion qRot1 = My.Quaternion.RotationQuaternion(Axis1, RotationAngle1);
            My.Quaternion qRot2 = My.Quaternion.RotationQuaternion(Axis2, RotationAngle2);

            DefaultRotation = qRot2 * qRot1;
        }

        /// <summary>
        /// Sets the default rotation for the RotateDefault() method.
        /// </summary>
        /// <param name="Axis1">A quaternion or a Vector3 object that represents the first axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle1">The angle of rotation around the first axis.</param>
        /// <param name="Axis2">A quaternion or a Vector3 object that represents the second axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle2">The angle of rotation around the second axis.</param>
        /// <param name="Axis3">A quaternion or a Vector3 object that represents the third axis. It is assumed
that the axis starts at the origin.</param>
        /// <param name="RotationAngle3">The angle of rotation around the third axis.</param>
        public void SetDefaultRotation(My.Quaternion Axis1, float RotationAngle1,
                                       My.Quaternion Axis2, float RotationAngle2,
                                       My.Quaternion Axis3, float RotationAngle3)
        {
            My.Quaternion qRot1 = My.Quaternion.RotationQuaternion(Axis1, RotationAngle1);
            My.Quaternion qRot2 = My.Quaternion.RotationQuaternion(Axis2, RotationAngle2);
            My.Quaternion qRot3 = My.Quaternion.RotationQuaternion(Axis3, RotationAngle3);

            DefaultRotation = qRot3 * qRot2 * qRot1;
        }

        /// <summary>
        /// Translates the shape over a specified vector.
        /// </summary>
        /// <param name="Translation">A quaternion or a Vector3 object that represents the translation.</param>
        public void Translate(My.Quaternion Translation)
        {
            for (int i = 0; i < currentVertices.Length; i++)
            {
                currentVertices[i] = new VertexPositionNormalTexture(
                    (Vector3)(currentVertices[i].Position + Translation),
                        Vector3.Up, currentVertices[i].TextureCoordinate);
            }
            Center = (Vector3)(Center + Translation);
        }

        /// <summary>
        /// Scales the shape.
        /// </summary>
        /// <param name="scale">the scaling coefficient.</param>
        public void Scale(float scale)
        {
            for (int i = 0; i < currentVertices.Length; i++)
            {
                currentVertices[i] = new VertexPositionNormalTexture(
                    (Vector3)((currentVertices[i].Position - Center) * scale + Center),
                        Vector3.Up, currentVertices[i].TextureCoordinate);
            }

        }

        /// <summary>
        /// Draws the shape to the specified GraphicsDevice.
        /// </summary>
        /// <param name="basicEffect">The effect of drawing.</param>
        /// <param name="mode">The drawing mode.</param>
        public void Draw(BasicEffect basicEffect, DrawMode mode)
        {
            basicEffect.TextureEnabled = ((mode & DrawMode.Textured) > 0);
            if (basicEffect.TextureEnabled)
                basicEffect.Texture = Texture;
```

```
            basicEffect.CurrentTechnique.Passes[0].Apply();

            if ((mode & DrawMode.Textured) > 0
                || (mode & DrawMode.Solid) > 0)
            {

GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionNormalTexture>(PrimitiveType.TriangleList,
currentVertices, 0, currentVertices.Length, triangleIndices, 0, triangleIndices.Length / 3,
VertexPositionNormalTexture.VertexDeclaration);
            }

            if ((mode & DrawMode.Lines) > 0)
            {
                GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionNormalTexture>(PrimitiveType.LineList,
currentVertices, 0, currentVertices.Length, lineIndices, 0, lineIndices.Length / 2,
VertexPositionNormalTexture.VertexDeclaration);
            }
        }
        #endregion

        #region Static methods
        /// <summary>
        /// Draws a set of shapes to the specified GraphicsDevice.
        /// The assumtion is that the camera is placed on the positive half of the Z axis.
        /// </summary>
        /// <param name="effect">The effect of drawing.</param>
        /// <param name="mode">The drawing mode.</param>
        /// <param name="shapes">The shapes to draw. The order they come is not important.</param>
        public static void DrawScene(BasicEffect effect, DrawMode mode, params Shape[] shapes)
        {
            float[] CenterZs = new float[shapes.Length];
            for (int i = 0; i < shapes.Length; i++)
                CenterZs[i] = shapes[i].Center.Z;
            Array.Sort(CenterZs, shapes);
            for (int i = 0; i < shapes.Length; i++)
                shapes[i].Draw(effect, mode);
        }

        /// <summary>
        /// Draws a set of shapes to the specified GraphicsDevice.
        /// The assumtion is that the camera is placed on the positive half of the Z axis.
        /// </summary>
        /// <param name="effect">The effect of drawing.</param>
        /// <param name="mode">The drawing mode.</param>
        /// <param name="shapes">The shapes to draw. The order they come is not important.</param>
        /*public static void DrawShapes(BasicEffect effect, DrawMode mode, Shape[] shapes)
        {
            float[] CenterZs = new float[shapes.Length];
            for (int i = 0; i < shapes.Length; i++)
                CenterZs[i] = shapes[i].Center.Z;
            Array.Sort(CenterZs, shapes);
            for (int i = 0; i < shapes.Length; i++)
                shapes[i].Draw(effect, mode);
        }*/
        /// <summary>
        /// Draws a set of shapes to the specified GraphicsDevice.
        /// The assumtion is that the camera is placed on the positive half of the Z axis.
        /// </summary>
        /// <param name="effect">The effect of drawing.</param>
        /// <param name="mode">The drawing mode.</param>
        /// <param name="shapes1">The shapes to draw. The order they come is not important.</param>
        /// <param name="shapes2">The shapes to draw. The order they come is not important.</param>
        public static void DrawScene(BasicEffect effect, DrawMode mode, Shape[] shapes1, params Shape[]
shapes2)
        {
            Shape[] shapes = new Shape[shapes1.Length+shapes2.Length];
            Array.Copy(shapes1, shapes, shapes1.Length);
            Array.Copy(shapes2, 0, shapes, shapes1.Length, shapes2.Length);

            float[] CenterZs = new float[shapes.Length];
            for (int i = 0; i < shapes.Length; i++)
                CenterZs[i] = shapes[i].Center.Z;
            Array.Sort(CenterZs, shapes);
            for (int i = 0; i < shapes.Length; i++)
                shapes[i].Draw(effect, mode);
        }
```

```
        #endregion
    }
}
```

2.1.2. *Модуль Ellipse.cs (Эллипс)*

```
using System;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace DCL.Phone.Xna
{
    /// <summary>
    /// Represents an ellipse which is actually a many sided polygon.
    /// </summary>
    public class Ellipse : Shape
    {
        #region Properties
        /// <summary>
        /// Gets the radius of the ellipse.
        /// </summary>
        public float Radius { get; private set; }

        /// <summary>
        /// Gets a normalized vector with direction from the center to the initial upper point.
        /// </summary>
        public Vector3 Axis
        {
            get
            {
                Vector3 v = new Vector3(currentVertices[0].Position.X,
                                        currentVertices[0].Position.Y,
                                        currentVertices[0].Position.Z) - Center;
                v.Normalize();
                return v;
            }
        }
        #endregion

        #region Constructors
        /// <summary>
        /// Sets up an ellipse.
        /// </summary>
        /// <param name="cent">The center of the ellipse.</param>
        /// <param name="radius">The radius of the ellipse.</param>
        /// <param name="radiusRatioX">The coefficient of stretching along the X axis.</param>
        /// <param name="radiusRatioY">The coefficient of stretching along the Y axis.</param>
        /// <param name="precision">A factor which influences the precision with that the ellipse is drawn. The
value is actually the number of sides in the polygon.</param>
        public Ellipse(Vector3 center, float radius, float radiusRatioX, float radiusRatioY, int precision)
        {
            if (radius <= 0) throw new ArgumentOutOfRangeException("radius", "Radius must be a positive
value");
            if (precision <= 0) throw new ArgumentOutOfRangeException("precision", "Precision coefficient must
be a positive value");

            Center = startCenter = center;
            Radius = radius;

            startVertices = new VertexPositionNormalTexture[precision * 2 + 1];
            currentVertices = new VertexPositionNormalTexture[precision * 2 + 1];
            lineIndices = new short[precision * 2];
            triangleIndices = new short[precision * 3];

            //SETTING UP AN ELLIPSE
```

```
            float t;//parameter
            for (int i = 0; i < precision; i++)
            {
                t = (float)Math.PI * 2 * i / precision;
                currentVertices[i * 2] = new VertexPositionNormalTexture
                                         (new Vector3(radius * radiusRatioX * (float)Math.Sin(t), radius *
radiusRatioY * (float)Math.Cos(t), 0) + Center,
                                         Vector3.Up, Vector2.Zero);
                t = (float)Math.PI * 2 * (i + 1) / precision;
                currentVertices[i * 2 + 1] = new VertexPositionNormalTexture
                                         (new Vector3(radius * radiusRatioX * (float)Math.Sin(t), radius *
radiusRatioY * (float)Math.Cos(t), 0) + Center,
                                         Vector3.Up, Vector2.UnitX);

                lineIndices[i * 2] = (short)(i * 2);
                lineIndices[i * 2 + 1] = (short)(i * 2 + 1);

                triangleIndices[i * 3] = (short)(i * 2);
                triangleIndices[i * 3 + 1] = (short)(i * 2 + 1);
                triangleIndices[i * 3 + 2] = (short)(precision * 2);
            }

            currentVertices[precision * 2] = new VertexPositionNormalTexture
                                         (Center, Vector3.Up, Vector2.UnitY);

            Array.Copy(currentVertices, startVertices, startVertices.Length);
        }

        /// <summary>
        /// Sets up an ellipse.
        /// </summary>
        /// <param name="cent">The center of the ellipse.</param>
        /// <param name="radius">The radius of the ellipse.</param>
        /// <param name="radiusRatioX">The coefficient of stretching along the X axis.</param>
        /// <param name="radiusRatioY">The coefficient of stretching along the Y axis.</param>
        /// <param name="precision">A factor which influences the precision with that the ellipse is drawn. The
value is actually the number of sides in the polygon.</param>
        /// <param name="texture">The texture that is drawn to the surface of the ellipse.</param>
        public Ellipse(Vector3 cent, float radius, float radiusRatioX, float radiusRatioY, int precision,
Texture2D texture) :
            this(cent, radius, radiusRatioX, radiusRatioY, precision)
        {
            Texture = texture;
        }

        /// <summary>
        /// Sets up an ellipse.
        /// </summary>
        /// <param name="cent">The center of the ellipse.</param>
        /// <param name="radius">The radius of the ellipse.</param>
        /// <param name="radiusRatioX">The coefficient of stretching along the X axis.</param>
        /// <param name="radiusRatioY">The coefficient of stretching along the Y axis.</param>
        /// <param name="precision">A factor which influences the precision with that the ellipse is drawn. The
value is actually the number of sides in the polygon.</param>
        /// <param name="texture">The texture that is drawn to the surface of the ellipse.</param>
        /// <param name="graphicsDevice">The graphics device to which the ellipse is drawn.</param>
        public Ellipse(Vector3 cent, float radius, float radiusRatioX, float radiusRatioY, int precision,
Texture2D texture, GraphicsDevice graphicsDevice) :
            this(cent, radius, radiusRatioX, radiusRatioY, precision, texture)
        {
            GraphicsDevice = graphicsDevice;
        }
        #endregion
    }
}
```

2.1.3. *Модуль Circle.cs (Окружность)*

```
using System;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
```

```
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace DCL.Phone.Xna
{
    /// <summary>
    /// Represents a circle which is actually a many sided polygon.
    /// </summary>
    public class Circle : Ellipse
    {
        #region Constructors
        /// <summary>
        /// Sets up a circle.
        /// </summary>
        /// <param name="center">The center of the circle.</param>
        /// <param name="radius">The radius of the circle.</param>
        /// <param name="precision">A factor which influences the precision with that the circle is drawn. The
value is actually the number of sides in the polygon.</param>
        public Circle(Vector3 center, float radius, int precision) : base(center, radius, 1, 1, precision){}

        /// <summary>
        /// Sets up a circle.
        /// </summary>
        /// <param name="cent">The center of the circle.</param>
        /// <param name="radius">The radius of the circle.</param>
        /// <param name="precision">A factor which influences the precision with that the circle is drawn. The
value is actually the number of sides in the polygon.</param>
        /// <param name="texture">The texture that is drawn to the surface of the circle.</param>
        public Circle(Vector3 cent, float radius, int precision, Texture2D texture):
            this(cent, radius, precision)
        {
            Texture = texture;
        }

        /// <summary>
        /// Sets up a circle.
        /// </summary>
        /// <param name="cent">The center of the circle.</param>
        /// <param name="radius">The radius of the circle.</param>
        /// <param name="precision">A factor which influences the precision with that the circle is drawn. The
value is actually the number of sides in the polygon.</param>
        /// <param name="texture">The texture that is drawn to the surface of the circle.</param>
        /// <param name="graphicsDevice">The graphics device to which the circle is drawn.</param>
        public Circle(Vector3 cent, float radius, int precision, Texture2D texture, GraphicsDevice
graphicsDevice):
            this(cent, radius, precision, texture)
        {
            GraphicsDevice = graphicsDevice;
        }
        #endregion
    }
}
```

2.1.4. Модуль Ellipsoid.cs (Эллипсоид)

```
using System;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace DCL.Phone.Xna
{
    /// <summary>
    /// Represents an ellipse which is actually a many sided three-dimensional polygon.
    /// </summary>
    public class Ellipsoid: Shape
    {
        #region Properties
        /// <summary>
        /// Gets the radius.
        /// </summary>
        public float Radius { get; private set; }

        /// <summary>
        /// Gets a normalized vector with direction from the center to the upper pole.
        /// </summary>
        public Vector3 Axis
        {
            get
            {
                Vector3 v = new Vector3(currentVertices[0].Position.X,
                                        currentVertices[0].Position.Y,
                                        currentVertices[0].Position.Z) - Center;
                v.Normalize();
                return v;
            }
        }
        #endregion

        #region Constructors
        /// <summary>
        /// Sets up an ellipse.
        /// </summary>
        /// <param name="cent">The center of the ellipse.</param>
        /// <param name="radius">The radius of the ellipse.</param>
        /// <param name="radiusRatioX">The coefficient of stretching along the X axis.</param>
        /// <param name="radiusRatioY">The coefficient of stretching along the Y axis.</param>
        /// <param name="radiusRatioZ">The coefficient of stretching along the Z axis.</param>
        /// <param name="precision">A factor which influences the precision with that the ellipse is drawn. The
recommended value for Windows Phone 7 is 10-14.</param>
        public Ellipsoid(Vector3 cent, float radius, float radiusRatioX, float radiusRatioY, float
radiusRatioZ, int precision)
        {
            if (radius <= 0) throw new ArgumentOutOfRangeException("radius", "Radius must be a positive
value");
            if (precision <= 0) throw new ArgumentOutOfRangeException("precision", "Precision coefficient must
be a positive value");

            Center = startCenter = cent;
            Radius = radius;
            precision *= 2;

            startVertices = new VertexPositionNormalTexture[precision * (precision + 1) * 4];
            currentVertices = new VertexPositionNormalTexture[precision * (precision + 1) * 4];
            lineIndices = new short[precision * precision * 8];
            triangleIndices = new short[precision * precision * 6];

            //SETTING UP A SPHERE
            float p, t; //parameters
            Vector3 Point; //Normal, temp;
            int ind;
```

```
            for (int j = 0; j < precision; j++)
            {//2 + (n-2)/2
                //SETTING UP A CIRCLE
                for (int i = 0; i < precision; i++)
                {
                    ind = (j * precision + i);

                    for (int m = 0; m < 2; m++)
                        for (int n = 0; n < 2; n++)
                        {
                            p = MathHelper.Pi * ((j + m) - precision / 2) / precision;
                            t = MathHelper.Pi * 2 * (i + n) / precision;
                            Point = new Vector3(radius * radiusRatioX * (float)(Math.Sin(t) * Math.Cos(p)),
                                                radius * radiusRatioY * (float)(Math.Cos(t)),
                                                radius * radiusRatioZ * (float)(Math.Sin(t) * Math.Sin(p)))
                                                + Center;

                            currentVertices[ind * 4 + n * 2 + m] = new VertexPositionNormalTexture
                                (Point, Vector3.Up,
                                (t / MathHelper.Pi < 1 || i == precision / 2 - 1) ? //!!!!!!!!!
                                new Vector2((MathHelper.PiOver2 - p) / MathHelper.TwoPi, t / MathHelper.Pi) :
                                new Vector2((MathHelper.PiOver2 - p) / MathHelper.TwoPi + 0.5f, 2 - t /
MathHelper.Pi));

                        }

                    lineIndices[ind * 8] = lineIndices[ind * 8 + 7] = (short)(ind * 4);
                    lineIndices[ind * 8 + 1] = lineIndices[ind * 8 + 2] = (short)(ind * 4 + 1);
                    lineIndices[ind * 8 + 6] = lineIndices[ind * 8 + 4] = (short)(ind * 4 + 2);
                    lineIndices[ind * 8 + 3] = lineIndices[ind * 8 + 5] = (short)(ind * 4 + 3);

                    if (i < precision / 2)
                    {
                        triangleIndices[ind * 6] = (short)(ind * 4);
                        triangleIndices[ind * 6 + 1] = triangleIndices[ind * 6 + 3] = (short)(ind * 4 + 2);
                        triangleIndices[ind * 6 + 2] = triangleIndices[ind * 6 + 5] = (short)(ind * 4 + 1);
                        triangleIndices[ind * 6 + 4] = (short)(ind * 4 + 3);
                    }
                    else
                    {
                        triangleIndices[ind * 6] = (short)(ind * 4);
                        triangleIndices[ind * 6 + 2] = triangleIndices[ind * 6 + 4] = (short)(ind * 4 + 2);
                        triangleIndices[ind * 6 + 1] = triangleIndices[ind * 6 + 5] = (short)(ind * 4 + 1);
                        triangleIndices[ind * 6 + 3] = (short)(ind * 4 + 3);
                    }
                }
            }

            Array.Copy(currentVertices, startVertices, startVertices.Length);
        }


        /// <summary>
        /// Sets up an ellipse.
        /// </summary>
        /// <param name="cent">The center of the ellipse.</param>
        /// <param name="radius">The radius of the ellipse.</param>
        /// <param name="radiusRatio">The coefficient of stretching along the X axis.</param>
        /// <param name="precision">A factor which influences the precision with that the ellipse is drawn. The
recommended value for Windows Phone 7 is 10-14.</param>
        /// <param name="texture">The texture that is drawn to the surface of the ellipse.</param>
        public Ellipsoid(Vector3 cent, float radius, float radiusRatioX, float radiusRatioY, float
radiusRatioZ, int precision, Texture2D texture) :
            this(cent, radius, radiusRatioX, radiusRatioY, radiusRatioZ, precision)
        {
            Texture = texture;
        }

        /// <summary>
        /// Sets up an ellipse.
        /// </summary>
        /// <param name="cent">The center of the ellipse.</param>
        /// <param name="radius">The radius of the ellipse.</param>
        /// <param name="radiusRatioX">The coefficient of stretching along the X axis.</param>
        /// <param name="radiusRatioY">The coefficient of stretching along the Y axis.</param>
        /// <param name="radiusRatioZ">The coefficient of stretching along the Z axis.</param>
```

```
        /// <param name="precision">A factor which influences the precision with that the ellipse is drawn. The
recommended value for Windows Phone 7 is 10-14.</param>
        /// <param name="texture">The texture that is drawn to the surface of the ellipse.</param>
        /// <param name="graphicsDevice">The graphics device to which the ellipse is drawn.</param>
        public Ellipsoid(Vector3 cent, float radius, float radiusRatioX, float radiusRatioY, float
radiusRatioZ, int precision, Texture2D texture, GraphicsDevice graphicsDevice) :
            this(cent, radius, radiusRatioX, radiusRatioY, radiusRatioZ, precision, texture)
        {
            GraphicsDevice = graphicsDevice;
        }
        #endregion
    }
}
```

2.1.5. Модуль Sphere.cs (Сфера)

```
using System;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace DCL.Phone.Xna
{
    /// <summary>
    /// Represents a sphere which is actually a many sided three-dimensional polygon.
    /// </summary>
    public class Sphere : Ellipsoid
    {
        #region Constructors
        /// <summary>
        /// Sets up a sphere.
        /// </summary>
        /// <param name="center">The center of the sphere.</param>
        /// <param name="radius">The radius of the sphere.</param>
        /// <param name="precision">A factor which influences the precision with that the sphere is drawn. The
recommended value for Windows Phone 7 is 10-14.</param>
        public Sphere(Vector3 center, float radius, int precision): base(center, radius, 1, 1, 1, precision) {}

        /// <summary>
        /// Sets up a sphere.
        /// </summary>
        /// <param name="cent">The center of the sphere.</param>
        /// <param name="radius">The radius of the sphere.</param>
        /// <param name="precision">A factor which influences the precision with that the sphere is drawn. The
recommended value for Windows Phone 7 is 10-14.</param>
        /// <param name="texture">The texture that is drawn to the surface of the sphere.</param>
        public Sphere(Vector3 center, float radius, int precision, Texture2D texture) :
            this(center, radius, precision)
        {
            Texture = texture;
        }

        /// <summary>
        /// Sets up a sphere.
        /// </summary>
        /// <param name="center">The center of the sphere.</param>
        /// <param name="radius">The radius of the sphere.</param>
        /// <param name="precision">A factor which influences the precision with that the sphere is drawn. The
recommended value for Windows Phone 7 is 10-14.</param>
        /// <param name="texture">The texture that is drawn to the surface of the sphere.</param>
        /// <param name="graphicsDevice">The graphics device to which the sphere is drawn.</param>
        public Sphere(Vector3 center, float radius, int precision, Texture2D texture, GraphicsDevice
graphicsDevice) :
            this(center, radius, precision, texture)
        {
            GraphicsDevice = graphicsDevice;
```

```
        }
        #endregion
    }
}
```

2.1.6. *Модуль Dot.cs (Точка)*

```
using System;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace DCL.Phone.Xna
{
    public class Dot: Shape
    {
        public Dot(Vector3 position, int scale)
        {
            if (scale <= 0) throw new ArgumentOutOfRangeException("scale", "Parameter must be grater than
zero");

            Center = startCenter = position;

            startVertices = new VertexPositionNormalTexture[3];
            currentVertices = new VertexPositionNormalTexture[3];
            lineIndices = new short[6];
            triangleIndices = new short[6];

            currentVertices[0] = new VertexPositionNormalTexture
                                    (position - 0.01f * scale * Vector3.UnitX + 0.01f * scale * Vector3.UnitZ,
                                    Vector3.Up, Vector2.Zero);
            currentVertices[1] = new VertexPositionNormalTexture
                                    (position + 0.01f * scale * Vector3.UnitX + 0.01f * scale * Vector3.UnitZ,
                                    Vector3.Up, Vector2.UnitX);
            currentVertices[2] = new VertexPositionNormalTexture
                                    (position - 0.01f * scale * Vector3.UnitZ,
                                    Vector3.Up, Vector2.UnitY);

            lineIndices[0] = 0; lineIndices[1] = 1; lineIndices[2] = 1; lineIndices[3] = 2; lineIndices[4] = 2;
lineIndices[5] = 0;
            triangleIndices[0] = 0; triangleIndices[1] = 1; triangleIndices[2] = 2; triangleIndices[3] = 2;
triangleIndices[4] = 1; triangleIndices[5] = 0;

            Array.Copy(currentVertices, startVertices, startVertices.Length);
        }

        public Dot(Vector3 position, int scale, Texture2D texture) :
            this(position, scale)
        {
            Texture = texture;
        }

        public Dot(Vector3 position, int scale, Texture2D texture, GraphicsDevice graphicsDevice) :
            this(position, scale, texture)
        {
            GraphicsDevice = graphicsDevice;
        }

    }
}
```

2.2 Классы, отвечающие за построение панорамного пользовательского интерфейса

2.2.1. Модуль PivotGame.cs (Основной класс)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Resources;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace DCL.Phone.Xna
{
    /// <summary>
    /// Provides an extending "wrapper" over the default Microsoft.Xna.Framework.Game
    /// with pivot functionality (pivot is a UI based on switching between tabs).
    /// XNA games that require the tab interface should derive from this class.
    /// </summary>
    public class PivotGame: Game
    {
        #region Fields
        int delta = 0; //Indicates the scrolling "distance"
        Vector2 deltaVector = Vector2.Zero; //For drawing texts
        int selInd = 0; //Selected tab index
        float cameraWidth=8, zoom=1; //Projection paramenters
        bool SelectedIndexSetFirstTime = true; //Selection changed event in not fired when loading the
pivot
        bool SwitchingTabsNow = false; //We are changing the selection right now
        ContentManager FontLoader; //For providing default fonts from the resource file
        List<int> CallStack = new List<int>(); //Is needed for handling the "Back" button
        ProjectionType proj; //Projection for BasicEffect

        /// <summary>
        /// The standard font for drawing page headers that can be changed.
        /// </summary>
        protected internal SpriteFont HeaderFont;
        /// <summary>
        /// The font that is used to draw text on the page and can be changed.
        /// </summary>
        protected internal SpriteFont ContentFont;
        /// <summary>
        /// The font that is used to draw the application title and can be changed
        /// (the recommended font size is 16-20)
        /// </summary>
        protected internal SpriteFont TitleFont;
        #endregion

        #region Events
        /// <summary>
        /// The devegate for the SelectionChanged event handler.
        /// </summary>
        /// <param name="sender">object that fires the event.</param>
        /// <param name="e">Event arguments that contain the previous selected page index.</param>
        public delegate void SelectionChangedEventHandler(object sender, SelectionChangedEventArgs e);
        /// <summary>
        /// Is fired when used changes the current pivot page by scrolling or tapping at the header.
        /// </summary>
        public event SelectionChangedEventHandler SelectionChanged;
        #endregion

        #region Properties
            #region Appearance
            /// <summary>
            /// Gets or sets the title of the application.
            /// </summary>
```

```
public string Title { get; set; }

/// <summary>
/// Gets or sets the background color of all the pivot pages.
/// </summary>
public Color BackgroundColor
{
    get { return this[SelectedIndex].BackgroundColor; }
    set
    {
        foreach (PivotGameItem pgi in Items)
            pgi.BackgroundColor = value;
    }
}

/// <summary>
/// Gets or sets the foreground (text) color of all the pivot pages.
/// </summary>
public Color ForegroundColor
{
    get { return this[SelectedIndex].ForegroundColor; }
                    set
{
    foreach (PivotGameItem pgi in Items)
        pgi.ForegroundColor = value;
}
}

/// <summary>
/// Gets or sets the background color of scenes on all the pivot pages.
/// </summary>
public Color SceneBackgroundColor
{
    get { return this[SelectedIndex].DrawingArea.BackgroundColor; }
    set
    {
        foreach (PivotGameItem pgi in Items)
            pgi.DrawingArea.BackgroundColor = value;
    }
}

/// <summary>
/// Gets or sets the background texture of scenes on all the pivot pages.
/// </summary>
public Texture2D SceneBackgroundTexture
{
    get { return this[SelectedIndex].DrawingArea.BackgroundTexture; }
    set
    {
        foreach (PivotGameItem pgi in Items)
            pgi.DrawingArea.BackgroundTexture = value;
    }
}

/// <summary>
/// Gets or sets the rectangle for the scenes on all the pivot pages.
/// </summary>
public Rectangle DrawingArea
{
    get
    {
        return new Rectangle(this[SelectedIndex].DrawingArea.X,
                             this[SelectedIndex].DrawingArea.Y,
                             this[SelectedIndex].DrawingArea.Width,
                             this[SelectedIndex].DrawingArea.Height);
    }
    set
    {
        foreach (PivotGameItem pgi in Items)
        {
            pgi.DrawingArea.X = value.X;
            pgi.DrawingArea.Y = value.Y;
            pgi.DrawingArea.Width = value.Width;
            pgi.DrawingArea.Height = value.Height;
            pgi.Update();
        }
```

```
            }
        }

        /// <summary>
        /// Gets or sets the image that is shown while loading the application.
        /// The property should be set in the LoadContent() method before calling base.LoadContent().
        /// </summary>
        public Texture2D SplashScreenImage { get; set; }
        #endregion

        #region Graphics
        /// <summary>
        /// Gets or sets the default GraphicsDeviceManager object of the game.
        /// </summary>
        protected internal GraphicsDeviceManager GraphicsDeviceManager { get; set; }

        /// <summary>
        /// Gets or sets the default SpriteBatch object that is used by the game to draw textures.
        /// </summary>
        protected internal SpriteBatch SpriteBatch { get; set; }

        /// <summary>
        /// Gets or sets the default BasicEffect object that is used by the game to draw 3D scenes.
        /// </summary>
        protected BasicEffect BasicEffect { get; set; }

        /// <summary>
        /// The projection type for BasicEffect.
        /// </summary>
        protected ProjectionType Projection
        {
            get { return proj; }
            set
            {
                proj = value;
                if (value == ProjectionType.Perspective)
                {
                    BasicEffect.View = Matrix.CreateLookAt(new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, CameraScale / Zoom), new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, 0), Vector3.Up);
                    BasicEffect.Projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90),
(float)GraphicsDeviceManager.PreferredBackBufferWidth / GraphicsDeviceManager.PreferredBackBufferHeight,
0.1f, 2 * CameraScale / Zoom);
                }
                else
                {
                    BasicEffect.View = Matrix.Identity;
                    BasicEffect.Projection = Matrix.CreateOrthographicOffCenter(-CameraScale / Zoom /
2 + SceneCenterTranslation.X,
                                                                CameraScale / Zoom / 2 +
SceneCenterTranslation.X,
                                                                -CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                -CameraScale, CameraScale);
                }
            }
        }

        /// <summary>
        /// Gets or sets the coefficient that is used to calculate the camera's width, height and
depth.
        /// </summary>
        public float CameraScale
        {
            get { return cameraWidth; }
            set
            {
                cameraWidth = value;

                if (Projection == ProjectionType.Perspective)
                {
```

```
                        BasicEffect.View = Matrix.CreateLookAt(new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, value / Zoom), new Vector3(SceneCenterTranslation.X, SceneCenterTranslation.Y,
0), Vector3.Up);
                        BasicEffect.Projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90),
(float)GraphicsDeviceManager.PreferredBackBufferWidth / GraphicsDeviceManager.PreferredBackBufferHeight,
0.1f, 2 * value / Zoom);
                    }
                    else
                    {
                        BasicEffect.View = Matrix.Identity;
                        BasicEffect.Projection = Matrix.CreateOrthographicOffCenter(-value / Zoom / 2 +
SceneCenterTranslation.X,
                                                                    value / Zoom / 2 + SceneCenterTranslation.X,
                                                                    -value *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                    value *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                    -value, value);
                    }
                }
            }

            /// <summary>
            /// Gets or sets the camera's zoom.
            /// </summary>
            public float Zoom
            {
                get { return zoom; }
                set
                {
                    zoom = value;

                    if (Projection == ProjectionType.Perspective)
                    {
                        BasicEffect.View = Matrix.CreateLookAt(new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, CameraScale / value), new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, 0), Vector3.Up);
                        BasicEffect.Projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90),
(float)GraphicsDeviceManager.PreferredBackBufferWidth / GraphicsDeviceManager.PreferredBackBufferHeight,
0.1f, 2 * CameraScale / value);
                    }
                    else
                    {
                        BasicEffect.View = Matrix.Identity;
                        BasicEffect.Projection = Matrix.CreateOrthographicOffCenter(-CameraScale / value /
2 + SceneCenterTranslation.X,
                                                                    CameraScale / value / 2 +
SceneCenterTranslation.X,
                                                                    -CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / value /
2 + SceneCenterTranslation.Y,
                                                                    CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / value /
2 + SceneCenterTranslation.Y,
                                                                    -CameraScale, CameraScale);
                    }
                }
            }

            /// <summary>
            /// Gets or sets the vector by that the 3D scene's center is translated.
            /// </summary>
            protected Vector2 SceneCenterTranslation { set; get; }

            internal int Delta
            {
                get { return delta; }
                set { delta = value; deltaVector = new Vector2(delta, 0); }
            }
            #endregion

            #region Pages
```

```
            //The pages
            private List<PivotGameItem> Items { get; set; }

            /// <summary>
            /// Gets or sets the current page index. If the index changes, then the SelectionChanged event
is fired.
            /// </summary>
            public int SelectedIndex
            {
                get { return selInd; }
                set
                {
                    int prev = selInd;
                    selInd = value;
                    if (!SelectedIndexSetFirstTime)
                    {
                        CallStack.Add(prev);
                        this[value].headerPosition = new Vector2(10, 40);
                        if (SelectionChanged != null && value != prev)
                            SelectionChanged(this, new SelectionChangedEventArgs(prev));

                        SwitchingTabsNow = true;
                        Delta = 480;
                        if ((prev > value && !(value == 0 && prev >= ItemsCount - 2)) || (prev < value &&
prev == 0 && value == ItemsCount - 1))
                            Delta = -480;
                    }
                    SelectedIndexSetFirstTime = false;
                }
            }

            /// <summary>
            /// The indexer that enables access to separate pivot pages.
            /// </summary>
            /// <param name="i">The index of page.</param>
            /// <returns>The PivotGameItem object that perpesents the page.</returns>
            public PivotGameItem this[int i]
            {
                get { return Items[i]; }
            }

            /// <summary>
            /// Gets the amount of pages in pivot.
            /// </summary>
            public int ItemsCount
            {
                get { return Items.Count; }
            }

            /// <summary>
            /// Indicates whether the pages are changing at the moment
            /// </summary>
            public bool ChangingPages
            {
                get { return Delta!=0;}
            }
            #endregion

            #region Touches
            /// <summary>
            /// Gets or sets the current state of the touch panel.
            /// It should be initialized before calling base.Update(gameTime).
            /// </summary>
            protected TouchCollection Touches { get; set; }
            #endregion
        #endregion

        #region Constructors
        /// <summary>
        /// The constructor which initializes the GraphicsDeviceManager object.
        /// </summary>
        public PivotGame()
        {
            GraphicsDeviceManager = new GraphicsDeviceManager(this);
            GraphicsDeviceManager.PreferredBackBufferWidth = 480;
            GraphicsDeviceManager.PreferredBackBufferHeight = 800;
```

```
            GraphicsDeviceManager.IsFullScreen = true;

            FontLoader = new ResourceContentManager(this.Services, DCL.Phone.Xna.Fonts.ResourceManager);
        }
        #endregion

        #region Pivot Methods

            /// <summary>
        /// Adds a new item to the pivot pages collection.
        /// </summary>
        /// <param name="pgi">The item to add.</param>
        public void AddItem(PivotGameItem pgi)
        {
            Items.Add(pgi);
            pgi.Parent = this;
        }

        //Used for handling the "Back" button.
        private int GetLastVisitedIndexFromCallStack()
        {
            if (CallStack.Count == 0) return -1;
            else
            {
                int i = CallStack[CallStack.Count - 1];
                CallStack.RemoveAt(CallStack.Count - 1);
                return i;
            }
        }
        #endregion

        #region Standard Methods

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content. Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            Items = new List<PivotGameItem>();
            AddItem(new PivotGameItem());

            selInd = 0;

            #region BasicEffect
            BasicEffect = new BasicEffect(GraphicsDevice);
            BasicEffect.LightingEnabled = true;
            BasicEffect.PreferPerPixelLighting = true;
            //BasicEffect.EnableDefaultLighting();
            BasicEffect.VertexColorEnabled = false;

            BasicEffect.AmbientLightColor = new Vector3(0.2f, 0.2f, 0.2f);

            BasicEffect.SpecularColor = new Vector3(1, 1, 1);
            BasicEffect.SpecularPower = 60;

            // Set direction of light here, not position!
            BasicEffect.DirectionalLight0.Direction = new Vector3(-1, -1, -1);
            BasicEffect.DirectionalLight0.DiffuseColor = new Vector3(1, 1, 1);
            BasicEffect.DirectionalLight0.SpecularColor = new Vector3(1, 1, 1);
            BasicEffect.DirectionalLight0.Enabled = true;

            BasicEffect.Alpha = 1;

            Projection = ProjectionType.Perspective;
            BasicEffect.View = Matrix.CreateLookAt(new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, CameraScale / Zoom), new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, 0), Vector3.Up);
            BasicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90),
(float)GraphicsDeviceManager.PreferredBackBufferWidth / GraphicsDeviceManager.PreferredBackBufferHeight,
0.1f, 2 * CameraScale / Zoom);
            #endregion

            base.Initialize();
```

```
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            #region Default fonts
            HeaderFont = FontLoader.Load<SpriteFont>("Segoe48Bold");
            TitleFont = FontLoader.Load<SpriteFont>("Segoe16Bold");
            ContentFont = FontLoader.Load<SpriteFont>("Segoe16");
            #endregion

            SpriteBatch = new SpriteBatch(GraphicsDevice);

            #region Splash screen
            if (SplashScreenImage != null)
            {
                SpriteBatch.Begin();
                SpriteBatch.Draw(SplashScreenImage, new Vector2(0, 0), Color.White);
                SpriteBatch.End();
                GraphicsDevice.Present();
            }
            #endregion
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// all content.
        /// </summary>
        protected override void UnloadContent()
        {
            #region BasicEffect
            if (BasicEffect != null)
            {
                BasicEffect.Dispose();
                BasicEffect = null;
            }
            #endregion
            #region SpriteBatch
            if (SpriteBatch != null)
            {
                SpriteBatch.Dispose();
                SpriteBatch = null;
            }
            #endregion
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            #region "Back" button
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            {
                int prev = GetLastVisitedIndexFromCallStack();
                if (prev == -1)
                    this.Exit();
                else
                {
                    int temp = selInd;
                    selInd = prev; //To avoid adding the item to the call stack
                    this[selInd].headerPosition = new Vector2(10, 40);
                    if (SelectionChanged != null)
                        SelectionChanged(this, new SelectionChangedEventArgs(temp));

                    SwitchingTabsNow = true;
                    Delta = -480;
                    if ((prev > temp && !(temp == 0 && prev >= ItemsCount - 2)) || (prev < temp && prev ==
0 && temp == ItemsCount - 1))
                        Delta = 480;
                }
```

```
            }
            #endregion

            #region Touches
            if (Touches.Count == 0) Touches = TouchPanel.GetState();
            if (Touches.Count == 1 && !SwitchingTabsNow)
            {
                #region Switching tabs
                if (Touches[0].State == TouchLocationState.Pressed &&
                        Touches[0].Position.Y >= this[SelectedIndex].headerPosition.Y &&
                        Touches[0].Position.Y <= this[SelectedIndex].headerPosition.Y +
HeaderFont.MeasureString(this[SelectedIndex].Header).Y)
                {
                    int ind = SelectedIndex;
                    float temp1, temp2 = this[ind].headerPosition.X +
HeaderFont.MeasureString(this[ind].Header).X + 10;

                    do
                    {
                        temp1 = temp2;
                        ind++;
                        if (ind == Items.Count) ind=0;
                        temp2 += HeaderFont.MeasureString(this[ind].Header).X + 10;
                        if (Touches[0].Position.X > temp1 && Touches[0].Position.X <= temp2)
                        {
                            SelectedIndex = ind;
                            break;
                        }
                    }while (temp1 < 480);
                }
                #endregion
                #region Moving finger beyond the scene area
                else if (Touches[0].State == TouchLocationState.Moved &&
                    !(Touches[0].Position.X - Delta > this[SelectedIndex].DrawingArea.X &&
                       Touches[0].Position.X - Delta < this[SelectedIndex].DrawingArea.X +
this[SelectedIndex].DrawingArea.Width &&
                       //Touches[0].Position.Y > this[SelectedIndex].DrawingArea.Y &&
                       Touches[0].Position.Y < this[SelectedIndex].DrawingArea.Y +
this[SelectedIndex].DrawingArea.Height))
                {
                    TouchLocation prevTouch;
                    if (Touches[0].TryGetPreviousLocation(out prevTouch))
                    {
                        Delta += (int)(Touches[0].Position.X - prevTouch.Position.X);
                        this[SelectedIndex].headerPosition = new Vector2
                                (10 + HeaderFont.MeasureString(this[SelectedIndex].Header).X * Delta /
480, 40);

                        if (Projection == ProjectionType.Perspective)
                        {
                            BasicEffect.View = Matrix.CreateLookAt(new Vector3(SceneCenterTranslation.X -
(float)Delta / 60, SceneCenterTranslation.Y, CameraScale / Zoom), new Vector3(SceneCenterTranslation.X -
(float)Delta / 60, SceneCenterTranslation.Y, 0), Vector3.Up);
                            BasicEffect.Projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90),
(float)GraphicsDeviceManager.PreferredBackBufferWidth / GraphicsDeviceManager.PreferredBackBufferHeight,
0.1f, 2 * CameraScale / Zoom);
                        }
                        else
                        {
                            BasicEffect.View = Matrix.Identity;
                            BasicEffect.Projection = Matrix.CreateOrthographicOffCenter(-CameraScale /
Zoom * (0.5f + (float)Delta / 480) + SceneCenterTranslation.X,
                                                                CameraScale / Zoom * (0.5f - (float)Delta / 480)
+ SceneCenterTranslation.X,
                                                                -CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                -CameraScale, CameraScale);
                        }
                    }
                }
                #endregion
```

```
                #region Releasing finger - we may have to switch tabs
                else if (Touches[0].State == TouchLocationState.Released)
                {
                    if (Delta > 100)
                    {
                        Delta = 0;

                        SelectedIndex = (SelectedIndex == 0) ? (ItemsCount - 1) : (SelectedIndex - 1);
                    }
                    else if (Delta < -100)
                    {
                        Delta = 0;

                        SelectedIndex = (SelectedIndex == ItemsCount-1) ? (0) : (SelectedIndex + 1);
                    }
                    else
                    {
                        Delta = 0;
                    }
                    this[SelectedIndex].headerPosition = new Vector2(10, 40);

                    if (Projection == ProjectionType.Perspective)
                    {
                        BasicEffect.View = Matrix.CreateLookAt(new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, CameraScale / Zoom), new Vector3(SceneCenterTranslation.X,
SceneCenterTranslation.Y, 0), Vector3.Up);
                        BasicEffect.Projection =
Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90),
(float)GraphicsDeviceManager.PreferredBackBufferWidth / GraphicsDeviceManager.PreferredBackBufferHeight,
0.1f, 2 * CameraScale / Zoom);
                    }
                    else
                    {
                        BasicEffect.View = Matrix.Identity;
                        BasicEffect.Projection = Matrix.CreateOrthographicOffCenter(-CameraScale / Zoom /
2 + SceneCenterTranslation.X,
                                                                    CameraScale / Zoom / 2 +
SceneCenterTranslation.X,
                                                                    -CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                    CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                                    -CameraScale, CameraScale);
                    }
                }
                #endregion
            }
            #endregion

            #region Switching tabs animation
            if (SwitchingTabsNow)
            {
                Delta += (Delta<0)?80:-80;
                if (Delta == 0) SwitchingTabsNow = false;

                if (Projection == ProjectionType.Perspective)
                {
                    BasicEffect.View = Matrix.CreateLookAt(new Vector3(SceneCenterTranslation.X -
(float)Delta / 60, SceneCenterTranslation.Y, CameraScale / Zoom), new Vector3(SceneCenterTranslation.X -
(float)Delta / 60, SceneCenterTranslation.Y, 0), Vector3.Up);
                    BasicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90),
(float)GraphicsDeviceManager.PreferredBackBufferWidth / GraphicsDeviceManager.PreferredBackBufferHeight,
0.1f, 2 * CameraScale / Zoom);
                }
                else
                {
                    BasicEffect.View = Matrix.Identity;
                    BasicEffect.Projection = Matrix.CreateOrthographicOffCenter(-CameraScale / Zoom *
(0.5f + (float)Delta / 480) + SceneCenterTranslation.X,
                                                                CameraScale / Zoom * (0.5f - (float)Delta / 480)
+ SceneCenterTranslation.X,
                                                                -CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
```

```
                                                        CameraScale *
GraphicsDeviceManager.PreferredBackBufferHeight / GraphicsDeviceManager.PreferredBackBufferWidth / Zoom /
2 + SceneCenterTranslation.Y,
                                                        -CameraScale, CameraScale);
                }
            }
            #endregion

            this[SelectedIndex].Update();

            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            Items[SelectedIndex].Draw();

            base.Draw(gameTime);
        }
        #endregion

        #region Other methods
        /// <summary>
        /// Draws a string (considers the possible page's displacement).
        /// </summary>
        /// <param name="font">Font of the text.</param>
        /// <param name="text">The text to draw</param>
        /// <param name="position">The position of the left upper corner of the text.</param>
        /// <param name="color">Color of the text.</param>
        public void DrawString(SpriteFont font, string text, Vector2 position, Color color)
        {
            SpriteBatch.DrawString(font, text, position + deltaVector, color);
        }

        /// <summary>
        /// Renders a texture (considers the possible page's displacement).
        /// </summary>
        /// <param name="sprite">The texture to draw.</param>
        /// <param name="position">The position of the left upper corner of the texture.</param>
        /// <param name="color">Color of the texture.</param>
        public void DrawSprite(Texture2D sprite, Vector2 position, Color color)
        {
            SpriteBatch.Draw(sprite, position + deltaVector, color);
        }
        #endregion
    }



    /// <summary>
    /// Arguments for the SelectionChanged event that contain the previous selected page index.
    /// </summary>
    public class SelectionChangedEventArgs : EventArgs
    {
        /// <summary>
        /// The index of the page that was current before the selection changed.
        /// </summary>
        public int PreviousIndex { get; set; }

        /// <summary>
        /// Constructor.
        /// </summary>
        /// <param name="previousIndex">The index of the page that was current before the selection
changed.</param>
        public SelectionChangedEventArgs(int previousIndex)
        {
            PreviousIndex = previousIndex;
        }
    }

    /// <summary>
    /// The projection type for BasicEffect
```

```
    /// </summary>
    public enum ProjectionType
    {
        /// <summary>
        /// Perspective projection.
        /// </summary>
        Perspective,

        /// <summary>
        /// Orthographic projection.
        /// </summary>
        Orthographic
    }
}
```

2.2.2. *Модуль PivotGameItem.cs (Страница приложения)*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Resources;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace DCL.Phone.Xna
{
    /// <summary>
    /// Represents single pivot pages that contain application title, tab header,
    /// a 3D scene and some additional graphics (e.g. text).
    /// A dynamic list of type PivotGameItem is stored in any class that derives from PivotGame.
    /// </summary>
    public class PivotGameItem
    {
        #region Fields
        /// <summary>
        /// The rectangle for a scene on the page.
        /// </summary>
        public DrawingArea DrawingArea;
        RenderTarget2D tinyTexture; //A texture of size 1x1 that can be colored in any way
        Rectangle rectDrawingArea, rectBackgr1, rectBackgr2, rectBackgr3, rectBackgr4; //Rectangles to
draw
        PivotGame pg; //Parent
        internal Vector2 headerPosition, titlePosition; //Text positions
        Color bkgColor, frgColor, deactivatedColor; //Rectangle colors
        float temp; //used in Draw() method
        int ind; //used in Draw() method
        #endregion

        #region Properties
        /// <summary>
        /// Gets or sets the background color of the page.
        /// </summary>
        public Color BackgroundColor
        {
            get { return bkgColor; }
            set
            {
                bkgColor = value;
                deactivatedColor = Color.FromNonPremultiplied((frgColor.R + 2*bkgColor.R) / 3,
                                        (frgColor.G + 2 * bkgColor.G) / 3, (frgColor.B + 2 *
bkgColor.B) / 3, 255);
            }
        }

        /// <summary>
        /// Gets or sets the foreground (text) color of the page.
```

```
        /// </summary>
        public Color ForegroundColor
        {
            get { return frgColor; }
            set
            {
                frgColor = value;
                deactivatedColor = Color.FromNonPremultiplied((frgColor.R + 2 * bkgColor.R) / 3,
                                                (frgColor.G + 2 * bkgColor.G) / 3, (frgColor.B + 2 *
bkgColor.B) / 3, 255);
            }
        }

        /// <summary>
        /// Gets or sets the tab header.
        /// </summary>
        public string Header { set; get; }

        /// <summary>
        /// Gets or sets the pivot the page belongs to.
        /// </summary>
        public PivotGame Parent
        {
            set
            {
                pg = value;
                tinyTexture = new RenderTarget2D(value.GraphicsDevice, 1, 1);
                value.GraphicsDevice.SetRenderTarget(tinyTexture);
                value.GraphicsDevice.Clear(Color.White);
                value.GraphicsDevice.SetRenderTarget(null);
            }
            get { return pg; }
        }
        #endregion

        #region Events
        /// <summary>
        /// An event that is fired each drawing cycle.
        /// Its handler should have the code for drawing
        /// the 3D scene.
        /// </summary>
        public event EventHandler DrawScene;

        /// <summary>
        /// An event that is fired each drawing cycle.
        /// Its handler should have the code for drawing
        /// additional graphics, e.g. texts.
        /// </summary>
        public event EventHandler DrawFrame;
        #endregion

        #region Constructors
        /// <summary>
        /// The page's constructor. Initializes the drawing area and the page colors by default values.
        /// </summary>
        public PivotGameItem()
        {
            DrawingArea.BackgroundColor = Color.DarkBlue;
            DrawingArea.X = 10;
            DrawingArea.Y = 200;
            DrawingArea.Width = 460;
            DrawingArea.Height = 300;

            headerPosition = new Vector2(10, 40);
            titlePosition = new Vector2(10, 0);

            BackgroundColor = Color.CornflowerBlue;
            ForegroundColor = Color.DarkGreen;
        }
        #endregion

        #region Methods
        //Updates the coordinates of the rectangle.
        internal void Update()
        {
            rectDrawingArea.X = DrawingArea.X + Parent.Delta;
```

```
            rectDrawingArea.Y = DrawingArea.Y;
            rectDrawingArea.Width = DrawingArea.Width;
            rectDrawingArea.Height = DrawingArea.Height;

            rectBackgr1.X = Parent.Delta;
            rectBackgr1.Y = 0;
            rectBackgr1.Width = 480;
            rectBackgr1.Height = rectDrawingArea.Y;

            rectBackgr2.X = Parent.Delta;
            rectBackgr2.Y = rectDrawingArea.Y;
            rectBackgr2.Width = rectDrawingArea.X - Parent.Delta;
            rectBackgr2.Height = rectDrawingArea.Height;

            rectBackgr3.X = rectDrawingArea.X + rectDrawingArea.Width;
            rectBackgr3.Y = rectDrawingArea.Y;
            rectBackgr3.Width = 480 - Parent.Delta - rectBackgr3.X;
            rectBackgr3.Height = rectDrawingArea.Height;

            rectBackgr4.X = Parent.Delta;
            rectBackgr4.Y = rectDrawingArea.Y + rectDrawingArea.Height;
            rectBackgr4.Width = 480;
            rectBackgr4.Height = 800 - rectBackgr4.Y;

        }

        //Is called from Draw(gameTime) methods from the PivotGame class.
        internal void Draw()
        {
            Parent.GraphicsDevice.Clear(BackgroundColor);

            Parent.SpriteBatch.Begin();
            if(DrawingArea.BackgroundTexture==null)
                Parent.SpriteBatch.Draw(tinyTexture, rectDrawingArea, DrawingArea.BackgroundColor);
            else
                Parent.SpriteBatch.Draw(DrawingArea.BackgroundTexture, rectDrawingArea, Color.White);
            Parent.SpriteBatch.End();

            if (DrawScene != null)
                DrawScene(this, EventArgs.Empty);
            Parent.SpriteBatch.Begin();
            Parent.SpriteBatch.Draw(tinyTexture, rectBackgr1, BackgroundColor);
            Parent.SpriteBatch.Draw(tinyTexture, rectBackgr2, BackgroundColor);
            Parent.SpriteBatch.Draw(tinyTexture, rectBackgr3, BackgroundColor);
            Parent.SpriteBatch.Draw(tinyTexture, rectBackgr4, BackgroundColor);

            Parent.SpriteBatch.DrawString(Parent.TitleFont, Parent.Title, titlePosition, ForegroundColor);
            Parent.SpriteBatch.DrawString(Parent.HeaderFont, Header, headerPosition, ForegroundColor);
            temp = headerPosition.X + Parent.HeaderFont.MeasureString(Header).X + 10;
            ind = Parent.SelectedIndex + 1;
            while (temp < 480)
            {
                if (ind == Parent.ItemsCount) ind = 0;
                Parent.SpriteBatch.DrawString(Parent.HeaderFont, Parent[ind].Header, new Vector2(temp,
headerPosition.Y), deactivatedColor);
                temp += Parent.HeaderFont.MeasureString(Parent[ind].Header).X + 10;
                ind++;
            }
            ind = (Parent.SelectedIndex == 0) ? (Parent.ItemsCount - 1) : (Parent.SelectedIndex - 1);
            Parent.SpriteBatch.DrawString(Parent.HeaderFont, Parent[ind].Header, new
Vector2(headerPosition.X - Parent.HeaderFont.MeasureString(Parent[ind].Header).X - 10, headerPosition.Y),
deactivatedColor);
            Parent.SpriteBatch.End();

            if (DrawFrame != null)
                DrawFrame(this, EventArgs.Empty);

        }
        #endregion
    }

    /// <summary>
    /// A structure that represents the rectangle in which the 3D scene is being drawn.
    /// </summary>
    public struct DrawingArea
    {
```

```
        /// <summary>
        /// The X coordinate of the left upper corner of the rectangle.
        /// </summary>
        public int X { get; set; }

        /// <summary>
        /// The Y coordinate of the left upper corner of the rectangle.
        /// </summary>
        public int Y { get; set; }

        /// <summary>
        /// The width of the rectangle.
        /// </summary>
        public int Width { get; set; }

        /// <summary>
        /// The height of the rectangle.
        /// </summary>
        public int Height { get; set; }

        /// <summary>
        /// The background color of the rectangle.
        /// </summary>
        public Color BackgroundColor { get; set; }

        /// <summary>
        /// The background texture of the rectangle.
        /// </summary>
        public Texture2D BackgroundTexture { get; set; }
    }
}
```

3. Программа Planets

3.1. Модуль Program.cs (Точка входа в программу)

```
using System;

namespace Planets
{
#if WINDOWS || XBOX
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Planets game = new Planets())
            {
                game.Run();
            }
        }
    }
#endif
}
```

3.2. Модуль TextContent.cs (Текстовые ресурсы)

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
```

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;
using System.Device.Location;
using My = DCL.Maths;
using DCL.Phone.Xna;

namespace Planets
{
    /// <summary>
    /// This is the main type of the application.
    /// Derives from DCL.Phone.Xna.PivotGame, which is
    /// an extending "wrapper" over the default Microsoft.Xna.Framework.Game
    /// with pivot functionality (UI based on switching between tabs).
    /// </summary>
    public partial class Planets : DCL.Phone.Xna.PivotGame
    {
        #region Textures
        Texture2D icoDiameter, icoMass, icoDistance, icoRotationPeriod, icoRevolutionPeriod,
icoGravitationalPull, icoTemperatures,
            icoMoon,
            icoPhobos, icoDeimos,
            icoEuropa, icoIo, icoCallisto, icoGanymede,
            icoTitan, icoDione, icoEnceladus, icoIapetus, icoTethys, icoMimas, icoRhea,
            icoTitania, icoOberon, icoAriel, icoUmbriel, icoMiranda,
            icoTriton, icoNereid, icoProteus,

            icoGPSon, icoGPSsearch, icoGPSoff;

            Vector2 posMoon = new Vector2(320, 630), posPhobos = new Vector2(320, 630), posDeimos = new
Vector2(320, 680),
                posIo = new Vector2(320, 630), posEuropa = new Vector2(320, 670), posGanymede = new
Vector2(320, 710),
                posCallisto = new Vector2(320, 750), posMimas = new Vector2(320, 630), posEnceladus = new
Vector2(320, 653),
                posTethys = new Vector2(320, 676), posDione = new Vector2(320, 699), posRhea = new
Vector2(320, 722),
                posTitan = new Vector2(320, 745), posIapetus = new Vector2(320, 768), posMiranda = new
Vector2(320, 630),
                posAriel = new Vector2(320, 662), posUmbriel = new Vector2(320, 694), posTitania = new
Vector2(320, 726),
                posOberon = new Vector2(320, 758), posProteus = new Vector2(320, 630), posTriton = new
Vector2(320, 670),
                posNereid = new Vector2(320, 710);
        #endregion

        #region Texts
            Vector2 coordPos1 = new Vector2(280, 2), coordPos2 = new Vector2(280, 18);
            Vector2 coordGps = new Vector2(440, 0);

            string strCharacteristics = "Characteristics:", strDiameter = "Diameter:", strMass = "Mass:",
strDistance = "Remoteness:",
                strRotationPeriod = "Day:", strRevolutionPeriod = "Year:",
                strGravitationalPull = "Gravitation:", strTemperatures = "Temp.:", strRemark = "*Measured
relative to the Earth",
                strMoons = "Moons:",

                strDot,

                mercuryDiameter, mercuryMass, mercuryDistance, mercuryRotationPeriod,
mercuryRevolutionPeriod, mercuryGravitationalPull, mercuryTemperatures, mercuryMoons,
                venusDiameter, venusMass, venusDistance, venusRotationPeriod, venusRevolutionPeriod,
venusGravitationalPull, venusTemperatures, venusMoons,
                earthDiameter, earthMass, earthDistance, earthRotationPeriod, earthRevolutionPeriod,
earthGravitationalPull, earthTemperatures, earthMoons,
                marsDiameter, marsMass, marsDistance, marsRotationPeriod, marsRevolutionPeriod,
marsGravitationalPull, marsTemperatures, marsMoons,
                jupiterDiameter, jupiterMass, jupiterDistance, jupiterRotationPeriod,
jupiterRevolutionPeriod, jupiterGravitationalPull, jupiterTemperatures, jupiterMoons,
                saturnDiameter, saturnMass, saturnDistance, saturnRotationPeriod, saturnRevolutionPeriod,
saturnGravitationalPull, saturnTemperatures, saturnMoons,
                uranusDiameter, uranusMass, uranusDistance, uranusRotationPeriod, uranusRevolutionPeriod,
uranusGravitationalPull, uranusTemperatures, uranusMoons,
                neptuneDiameter, neptuneMass, neptuneDistance, neptuneRotationPeriod,
neptuneRevolutionPeriod, neptuneGravitationalPull, neptuneTemperatures, neptuneMoons,
```

```
                strMoon = "Moon", strPhobos = "Phobos", strDeimos = "Deimos", strIo = "Io", strEuropa =
"Europa",
                strGanymede = "Ganymede", strCallisto = "Callisto", strTitan = "Titan", strDione =
"Dione",
                strEnceladus = "Enceladus", strIapetus = "Iapetus", strTethys = "Tethys", strMimas =
"Mimas", strRhea = "Rhea",
                strTitania = "Titania", strOberon = "Oberon", strAriel = "Ariel", strUmbriel = "Umbriel",
                strMiranda = "Miranda", strProteus = "Proteus", strTriton = "Triton", strNereid =
"Nereid",

                strBuy = "Buy now!";

            Vector2 coordCharacteristics = new Vector2(5, 600), coordDiameter = new Vector2(30, 625),
coordMass = new Vector2(30, 647),
                coordDistance = new Vector2(30, 669), coordRotationPeriod = new Vector2(30, 691),
coordRevolutionPeriod = new Vector2(30, 713),
                coordTemperatures = new Vector2(30, 735), coordGravitationalPull = new Vector2(30, 757),
coordNotation = new Vector2(10, 787),
                coordMoons = new Vector2(320, 600),

                spDiameter = new Vector2(7, 631), spMass = new Vector2(7, 653),
                spDistance = new Vector2(7, 675), spRotationPeriod = new Vector2(7, 697),
spRevolutionPeriod = new Vector2(7, 719),
                spTemperatures = new Vector2(7, 741), spGravitationalPull = new Vector2(7, 763),

                plDiameter = new Vector2(155, 625), plMass = new Vector2(100, 647),
                plDistance = new Vector2(178, 669), plRotationPeriod = new Vector2(85, 691),
plRevolutionPeriod = new Vector2(100, 713),
                plTemperatures = new Vector2(110, 735), plGravitationalPull = new Vector2(190, 757),
                plMoons = new Vector2(415, 600),

                coordMoon = new Vector2(390, 640), coordPhobos = new Vector2(380, 640), coordDeimos = new
Vector2(380, 690),
                coordIo = new Vector2(370, 635), coordEuropa = new Vector2(370, 675), coordGanymede = new
Vector2(370, 715),
                coordCallisto = new Vector2(370, 755), coordMimas = new Vector2(355, 627), coordEnceladus
= new Vector2(355, 650),
                coordTethys = new Vector2(355, 673), coordDione = new Vector2(355, 696), coordRhea = new
Vector2(355, 719),
                coordTitan = new Vector2(355, 742), coordIapetus = new Vector2(355, 765), coordMiranda =
new Vector2(362, 630),
                coordAriel = new Vector2(362, 662), coordUmbriel = new Vector2(362, 694), coordTitania =
new Vector2(362, 726),
                coordOberon = new Vector2(362, 758), coordProteus = new Vector2(370, 635), coordTriton =
new Vector2(370, 675),
                coordNereid = new Vector2(370, 715),

                coordBuy = new Vector2(320, 10);
        #endregion


        void LoadTextContent()
        {
            strDot = System.Globalization.NumberFormatInfo.CurrentInfo.NumberDecimalSeparator;

            mercuryDiameter = "0" + strDot + "382";
            mercuryMass = "0" + strDot + "06";
            mercuryDistance = "0" + strDot + "39";
            mercuryRotationPeriod = "58" + strDot + "64";
            mercuryRevolutionPeriod = "0" + strDot + "24";
            mercuryTemperatures = "-290/810 F°";
            mercuryGravitationalPull = "0" + strDot + "38";
            mercuryMoons = "0";

            venusDiameter = "0" + strDot + "949";
            venusMass = "0" + strDot + "82";
            venusDistance = "0" + strDot + "72";
            venusRotationPeriod = "243" + strDot + "02";
            venusRevolutionPeriod = "0" + strDot + "62";
            venusTemperatures = "864 F°";
            venusGravitationalPull = "0" + strDot + "91";
            venusMoons = "0";

            earthDiameter = "1" + strDot + "00";
            earthMass = "1" + strDot + "00";
```

```
        earthDistance = "1" + strDot + "00";
        earthRotationPeriod = "1" + strDot + "00";
        earthRevolutionPeriod = "1" + strDot + "00";
        earthTemperatures = "-128/136 F°";
        earthGravitationalPull = "1" + strDot + "00";
        earthMoons = "1";

        marsDiameter = "0" + strDot + "532";
        marsMass = "0" + strDot + "11";
        marsDistance = "1" + strDot + "52";
        marsRotationPeriod = "1" + strDot + "03";
        marsRevolutionPeriod = "1" + strDot + "88";
        marsTemperatures = "-189/63 F°";
        marsGravitationalPull = "0" + strDot + "38";
        marsMoons = "2";

        jupiterDiameter = "11" + strDot + "209";
        jupiterMass = "317" + strDot + "80";
        jupiterDistance = "5" + strDot + "20";
        jupiterRotationPeriod = "0" + strDot + "41";
        jupiterRevolutionPeriod = "11" + strDot + "86";
        jupiterTemperatures = "-193/63 F°";
        jupiterGravitationalPull = "2" + strDot + "54";
        jupiterMoons = "63";

        saturnDiameter = "9" + strDot + "449";
        saturnMass = "95" + strDot + "20";
        saturnDistance = "9" + strDot + "54";
        saturnRotationPeriod = "0" + strDot + "43";
        saturnRevolutionPeriod = "29" + strDot + "46";
        saturnTemperatures = "-285 F°";
        saturnGravitationalPull = "0" + strDot + "93";
        saturnMoons = "62";

        uranusDiameter = "4" + strDot + "007";
        uranusMass = "14" + strDot + "60";
        uranusDistance = "19" + strDot + "22";
        uranusRotationPeriod = "0" + strDot + "72";
        uranusRevolutionPeriod = "84" + strDot + "01";
        uranusTemperatures = "-357 F°";
        uranusGravitationalPull = "0" + strDot + "80";
        uranusMoons = "27";


        neptuneDiameter = "3" + strDot + "883";
        neptuneMass = "17" + strDot + "20";
        neptuneDistance = "30" + strDot + "06";
        neptuneRotationPeriod = "0" + strDot + "67";
        neptuneRevolutionPeriod = "164" + strDot + "80";
        neptuneTemperatures = "-360 F°";
        neptuneGravitationalPull = "1" + strDot + "20";
        neptuneMoons = "13";
    }

    void LoadIcons()
    {
        icoGPSon = Content.Load<Texture2D>("GPS_on");
        icoGPSoff = Content.Load<Texture2D>("GPS_off");
        icoGPSsearch = Content.Load<Texture2D>("GPS_search");
        icoDiameter = Content.Load<Texture2D>("diameter");
        icoMass = Content.Load<Texture2D>("mass");
        icoDistance = Content.Load<Texture2D>("remoteness");
        icoRotationPeriod = Content.Load<Texture2D>("day");
        icoRevolutionPeriod = Content.Load<Texture2D>("year");
        icoGravitationalPull = Content.Load<Texture2D>("gravitation");
        icoTemperatures = Content.Load<Texture2D>("temperatures");

        icoMoon = Content.Load<Texture2D>("ico_moon");
        icoPhobos = Content.Load<Texture2D>("ico_phobos");
        icoDeimos = Content.Load<Texture2D>("ico_deimos");
        icoIo = Content.Load<Texture2D>("ico_io");
        icoEuropa = Content.Load<Texture2D>("ico_europa");
        icoGanymede = Content.Load<Texture2D>("ico_ganymede");
        icoCallisto = Content.Load<Texture2D>("ico_callisto");
        icoTitan = Content.Load<Texture2D>("ico_titan");
        icoDione = Content.Load<Texture2D>("ico_dione");
```

```
            icoEnceladus = Content.Load<Texture2D>("ico_enceladus");
            icoIapetus = Content.Load<Texture2D>("ico_iapetus");
            icoTethys  = Content.Load<Texture2D>("ico_tethys");
            icoMimas = Content.Load<Texture2D>("ico_mimas");
            icoRhea = Content.Load<Texture2D>("ico_rhea");
            icoTitania = Content.Load<Texture2D>("ico_titania");
            icoOberon = Content.Load<Texture2D>("ico_oberon");
            icoAriel = Content.Load<Texture2D>("ico_ariel");
            icoUmbriel = Content.Load<Texture2D>("ico_umbriel");
            icoMiranda = Content.Load<Texture2D>("ico_miranda");
            icoTriton = Content.Load<Texture2D>("ico_triton");
            icoNereid = Content.Load<Texture2D>("ico_nereid");
            icoProteus = Content.Load<Texture2D>("ico_proteus");
        }
    }
}
```

3.3. *Модуль RingSector.cs (Построение кольцевого сектора)*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DCL.Phone.Xna;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace Planets
{
    class FlatRingSector: Shape
    {
        public float Radius { get; private set; }

        public float Width { get; private set; }

        public FlatRingSector(Vector3 center, float radius, float width, int precision, float angle)
        {
            Center = center;
            Radius = radius;

            startVertices = new VertexPositionNormalTexture[precision * 4];
            currentVertices = new VertexPositionNormalTexture[precision * 4];
            lineIndices = new short[precision * 8];
            triangleIndices = new short[precision * 12];

            //SETTING UP A RING
            float t;
            for (int i = 0; i < precision; i++)
            {
                t = angle * i / precision;
                currentVertices[i * 4] = new VertexPositionNormalTexture
                                         (new Vector3((radius + width / 2) * (float)Math.Sin(t), 0, (radius + width
/ 2) * (float)Math.Cos(t)) + Center,
                                         Vector3.Up, new Vector2((float)i/precision, 0));
                currentVertices[i * 4 + 3] = new VertexPositionNormalTexture
                                         (new Vector3((radius - width / 2) * (float)Math.Sin(t), 0, (radius - width
/ 2) * (float)Math.Cos(t)) + Center,
                                         Vector3.Up, new Vector2((float)i / precision, 1));

                t = angle * (i + 1) / precision;
                currentVertices[i * 4 + 1] = new VertexPositionNormalTexture
                                         (new Vector3((radius + width / 2) * (float)Math.Sin(t), 0,  (radius + width
/ 2) * (float)Math.Cos(t)) + Center,
                                         Vector3.Up, new Vector2((float)(i + 1) / precision, 0));
                currentVertices[i * 4 + 2] = new VertexPositionNormalTexture
```

```
                                    (new Vector3((radius - width / 2) * (float)Math.Sin(t), 0, (radius - width
/ 2) * (float)Math.Cos(t)) + Center,
                                    Vector3.Up, new Vector2((float)(i + 1) / precision, 1));

                lineIndices[i * 8] = (short)(i * 4);
                lineIndices[i * 8 + 1] = (short)(i * 4 + 1);
                lineIndices[i * 8 + 2] = (short)(i * 4 + 1);
                lineIndices[i * 8 + 3] = (short)(i * 4 + 2);
                lineIndices[i * 8 + 4] = (short)(i * 4 + 2);
                lineIndices[i * 8 + 5] = (short)(i * 4 + 3);
                lineIndices[i * 8 + 6] = (short)(i * 4 + 3);
                lineIndices[i * 8 + 7] = (short)(i * 4);

                //Кольцо - с обейих сторон
                triangleIndices[i * 12] = (short)(i * 4);
                triangleIndices[i * 12 + 1] = (short)(i * 4 + 1);
                triangleIndices[i * 12 + 2] = (short)(i * 4 + 2);

                triangleIndices[i * 12 + 3] = (short)(i * 4 + 2);
                triangleIndices[i * 12 + 4] = (short)(i * 4 + 3);
                triangleIndices[i * 12 + 5] = (short)(i * 4);

                triangleIndices[i * 12 + 6] = (short)(i * 4);
                triangleIndices[i * 12 + 7] = (short)(i * 4 + 2);
                triangleIndices[i * 12 + 8] = (short)(i * 4 + 1);

                triangleIndices[i * 12 + 9] = (short)(i * 4 + 3);
                triangleIndices[i * 12 + 10] = (short)(i * 4 + 2);
                triangleIndices[i * 12 + 11] = (short)(i * 4);
            }

            Center = currentVertices[2].Position;
            startCenter = Center;

            Array.Copy(currentVertices, startVertices, startVertices.Length);

        }

        public FlatRingSector(Vector3 center, float radius, float width, int precision, float angle, Texture2D
texture) :
            this(center, radius, width, precision, angle)
        {
            Texture = texture;
        }

        public FlatRingSector(Vector3 center, float radius, float width, int precision, float angle, Texture2D
texture, GraphicsDevice graphicsDevice) :
            this(center, radius, width, precision, angle, texture)
        {
            GraphicsDevice = graphicsDevice;
        }
    }
}
```

3.4. *Модуль FPS.cs (Компонент тестирования, отображающий значение FPS)*

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;

namespace Planets
{
    /// <summary>
```

```
    /// A reusable FPS-counter component, written by Shawn Hargreaves and modified for WP7 by Mikhail
Dubov.
    /// To use it, add "Components.Add(new FrameRateCounter(this));" to the game class constructor.
    /// </summary>
    public class FrameRateCounter : DrawableGameComponent
    {
        ContentManager content;
        SpriteBatch spriteBatch;
        SpriteFont spriteFont;

        int frameRate = 0;
        int frameCounter = 0;
        TimeSpan elapsedTime = TimeSpan.Zero;


        public FrameRateCounter(Game game)
            : base(game)
        {
            content = new ContentManager(game.Services);
            content.RootDirectory = "Content";
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
            spriteFont = content.Load<SpriteFont>("Segoe16");
        }


        protected override void UnloadContent()
        {
            content.Unload();
        }


        public override void Update(GameTime gameTime)
        {
            elapsedTime += gameTime.ElapsedGameTime;

            if (elapsedTime > TimeSpan.FromSeconds(1))
            {
                elapsedTime -= TimeSpan.FromSeconds(1);
                frameRate = frameCounter;
                frameCounter = 0;
            }
        }


        public override void Draw(GameTime gameTime)
        {
            frameCounter++;

            string fps = string.Format("fps: {0}", frameRate);

            spriteBatch.Begin();

            spriteBatch.DrawString(spriteFont, fps, new Vector2(20, 23), Color.White);

            spriteBatch.End();
        }
    }
}
```

3.5. *Модуль Settings.cs (Сохранение/восстановление состояния приложения)*

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
```

```
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;
using System.IO.IsolatedStorage;

namespace Planets
{
    public partial class Planets : DCL.Phone.Xna.PivotGame
    {
        //Loading settings
        protected override void OnActivated(object sender, EventArgs args)
        {
            if(!SettingsLoaded) LoadSettings();

            base.OnActivated(sender, args);
        }

        //Saving settings
        protected override void OnDeactivated(object sender, EventArgs args)
        {
            SaveSettings();

            base.OnDeactivated(sender, args);
        }

        void LoadSettings()
        {
            IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;

            if (!settings.TryGetValue<float>("Latitude", out Latitude))
                Latitude = 181;
            if (!settings.TryGetValue<float>("Longitude", out Longitude))
                Longitude = 181;
            if (!settings.TryGetValue<bool>("UseGPS", out UseGPS))
                Guide.BeginShowMessageBox("Privacy statement", "This application makes use of the built-in
location services.\n\nYour location will be used ONLY to indicate your position on the globe.\n\nEnable
the access to and use of location from the location services?", new string[] { "Enable", "Disable" }, 0,
MessageBoxIcon.Alert, new AsyncCallback(OnGPSSettingsClosed), null);

            //Start connecting GPS
            if (UseGPS) InitWatcher();
            else gpsState = 0;

            SettingsLoaded = true;
        }

        void SaveSettings()
        {
            IsolatedStorageSettings settings = IsolatedStorageSettings.ApplicationSettings;

            settings["Latitude"] = Latitude;
            settings["Longitude"] = Longitude;
            settings["UseGPS"] = UseGPS;
            settings.Save();
        }
    }
}
```

3.6. *Модуль Location.cs (Определение и визуализация местоположения устройства)*

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
```

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;
using System.Device.Location;
using DCL.Maths;
using DCL.Phone.Xna;

namespace Planets
{
    public partial class Planets : DCL.Phone.Xna.PivotGame
    {
        #region Coordinates
        float Latitude=181, Longitude=181;
        #endregion


        void InitWatcher()
        {
            watcher = new GeoCoordinateWatcher(GeoPositionAccuracy.Default);
            watcher.MovementThreshold = 20;

            watcher.StatusChanged += new
EventHandler<GeoPositionStatusChangedEventArgs>(watcher_StatusChanged);
            watcher.PositionChanged += new
EventHandler<GeoPositionChangedEventArgs<GeoCoordinate>>(watcher_PositionChanged);

            watcher.Start();
        }

        void UpdatePositionVisualisation()
        {
            tLocation1 = "Lat: " + (Angle)Latitude;
            tLocation2 = "Lon: " + (Angle)Longitude;
            RenderTarget2D rt = new RenderTarget2D(GraphicsDevice, tEarth.Width, tEarth.Height);
            GraphicsDevice.SetRenderTarget(rt);
            SpriteBatch.Begin();
            SpriteBatch.Draw(tEarth, Vector2.Zero, Color.White);
                SpriteBatch.DrawString(ContentFont, ".", new Vector2((Longitude < 0 ? Longitude + 360 :
Longitude) / 360 * tEarth.Width,
                                                                     tEarth.Height/2 - Latitude / 180 *
tEarth.Height-20), Color.Red);
            SpriteBatch.End();
            GraphicsDevice.SetRenderTarget(null);
            if(!IsTrial) Planet.Texture = (Texture2D) rt;
        }


        void watcher_StatusChanged(object sender, GeoPositionStatusChangedEventArgs e)
        {
            switch (e.Status)
            {
                case GeoPositionStatus.Disabled:
                case GeoPositionStatus.NoData:
                    gpsState = 0;
                    watcher.Stop();
                    if (Longitude <= 180)
                        UpdatePositionVisualisation();
                    break;

                case GeoPositionStatus.Initializing:
                    gpsState = 1;
                    break;

                case GeoPositionStatus.Ready:
                    gpsState = 2;
                    if (watcher.Position.Location.Longitude <= 180)
                    {
                        Latitude = (float)watcher.Position.Location.Latitude;
                        Longitude = (float)watcher.Position.Location.Longitude;
                        watcher.Stop();
                        UpdatePositionVisualisation();
                    }
                    break;
            }
        }
```

```
        void watcher_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
        {
            if (watcher.Position.Location.Longitude <= 180)
            {
                Latitude = (float)watcher.Position.Location.Latitude;
                Longitude = (float)watcher.Position.Location.Longitude;
                watcher.Stop();
                UpdatePositionVisualisation();
            }
        }

        private void OnGPSSettingsClosed(IAsyncResult ar)
        {
            switch (Guide.EndShowMessageBox(ar))
            {
                case 0:
                    UseGPS = true;
                    InitWatcher();
                    break;

                case 1:
                    UseGPS = false;
                    gpsState = 0;
                    tLocation1 = tLocation2 = "";
                    Planet.Texture = tEarth;
                    break;
            }
        }
        private void OnBuyMessageClosed(IAsyncResult ar)
        {
            if (Guide.EndShowMessageBox(ar) == 0)
                Guide.ShowMarketplace(PlayerIndex.One);
        }
    }
}
```

3.7 Модуль Planets.cs (Основная логика программы)

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Input.Touch;
using Microsoft.Xna.Framework.Media;
using System.Device.Location;
using My = DCL.Maths;
using DCL.Phone.Xna;

namespace Planets
{
    /// <summary>
    /// This is the main type of the application.
    /// Derives from DCL.Phone.Xna.PivotGame, which is
    /// an extending "wrapper" over the default Microsoft.Xna.Framework.Game
    /// with pivot functionality (UI based on switching between tabs).
    /// </summary>
    public partial class Planets : DCL.Phone.Xna.PivotGame
    {
        #region Fields

        #region Shapes
        Ellipsoid Planet, Moon, Moon2, Moon3, Moon4, Moon5, Moon6, Moon7;
        FlatRingSector[] SaturnRing = new FlatRingSector[100];
        #endregion
```

```
#region Rotation&Quaternions
My.Quaternion qAxis = My.Quaternion.j, qNew;
float RotationAngle = 0;
double touchesDistance;
Random randomAngle = new Random();
Vector3 MoonAxis;
#endregion

#region Textures
Texture2D tMercury, tVenus, tEarth, tMoon, tMars, tJupiter, tSaturn, tUranus, tNeptune,
    tPhobos, tDeimos,
    tEuropa, tIo, tCallisto, tGanymede,
    tSaturnRing, tTitan, tDione, tEnceladus, tIapetus, tTethys, tMimas, tRhea,
    tTitania, tOberon, tAriel, tUmbriel, tMiranda,
    tTriton, tNereid, tProteus;
#endregion

#region Location
GeoCoordinateWatcher watcher;
string tLocation1 = "", tLocation2 = "";
int gpsState = 1;
bool UseGPS;
bool SettingsLoaded = false;
#endregion

#region Text
SpriteFont tinyFont, smallFont;
#endregion

bool IsTrial;

#endregion

#region Constants
const int PRECISEMENT = 18;

//The angle between the equator plane and the perpendicular to the orbit is 23° 30'
//Obliquities that are less than 5 degrees are ignored
const float EARTH_OBLIQUITY = 0.409f;
const float MOON_EARTH_OBLIQUITY = 0.09f;

//All other speeds depend on this value
const float EARTH_ROTATION_SPEED = 0.003f;
//The moon rotates arund the Earth in 27.3 days
const float MOON_ROTATION_SPEED = EARTH_ROTATION_SPEED/27.3f;

//Actually mercury's solat day lasts 176 earth's days
const float MERCURY_ROTATION_SPEED = EARTH_ROTATION_SPEED / 10;

//Venera rotates in the opposite direction; Actually it's solar day lasts 117 earth's days
const float VENUS_ROTATION_SPEED = -EARTH_ROTATION_SPEED / 7;

//Mars' day is nearly the same as on the earth
const float MARS_ROTATION_SPEED = EARTH_ROTATION_SPEED;
const float PHOBOS_ROTATION_SPEED = MARS_ROTATION_SPEED * 24 / 7.6f;
const float DEIMOS_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.26f;
const float MARS_OBLIQUITY = 0.44f;

const float JUPITER_ROTATION_SPEED = EARTH_ROTATION_SPEED / 2.4f;
const float EUROPA_ROTATION_SPEED = EARTH_ROTATION_SPEED / 3.55f;
const float IO_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.77f;
const float GANYMEDE_ROTATION_SPEED = EARTH_ROTATION_SPEED / 7.15f;
const float CALLISTO_ROTATION_SPEED = EARTH_ROTATION_SPEED / 16.69f;

const float SATURN_ROTATION_SPEED = EARTH_ROTATION_SPEED / 2.2f;
const float TITAN_ROTATION_SPEED = EARTH_ROTATION_SPEED / 15.94f;
const float TETHYS_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.89f;
const float RHEA_ROTATION_SPEED = EARTH_ROTATION_SPEED / 4.5f;
const float MIMAS_ROTATION_SPEED = EARTH_ROTATION_SPEED / 0.94f;
const float IAPETUS_ROTATION_SPEED = EARTH_ROTATION_SPEED / 79.32f;
const float IAPETUS_SATURN_OBLIQUITY = 0.27f;
const float ENCELADUS_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.37f;
const float DIONE_ROTATION_SPEED = EARTH_ROTATION_SPEED / 2.77f;
const float SATURN_OBLIQUITY = 0.49f;

const float URANUS_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.4f;
```

```
const float TITANIA_ROTATION_SPEED = EARTH_ROTATION_SPEED / 8.71f;
const float OBERON_ROTATION_SPEED = EARTH_ROTATION_SPEED / 13.76f;
const float ARIEL_ROTATION_SPEED = EARTH_ROTATION_SPEED / 2.52f;
const float UMBRIEL_ROTATION_SPEED = EARTH_ROTATION_SPEED / 4.14f;
const float MIRANDA_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.41f;
const float URANUS_OBLIQUITY = 1.7f;

const float NEPTUNE_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.5f;
const float TRITON_ROTATION_SPEED = EARTH_ROTATION_SPEED / 5.88f;
const float TRITON_NEPTUNE_OBLIQUITY = 2.74f; //Triton rotates in the opposite direction
const float NEREID_ROTATION_SPEED = EARTH_ROTATION_SPEED / 360.13f;
const float PROTEUS_ROTATION_SPEED = EARTH_ROTATION_SPEED / 1.12f;
const float NEPTUNE_OBLIQUITY = 0.52f;
#endregion

//Constructor
public Planets()
{
    Content.RootDirectory = "Content";

    // Frame rate is 30 fps by default for Windows Phone.
    TargetElapsedTime = TimeSpan.FromTicks(333333);

    //Components.Add(new FrameRateCounter(this));
    //Guide.SimulateTrialMode = true;
}

/// <summary>
/// Allows the game to perform any initialization it needs to before starting to run.
/// This is where it can query for any required services and load any non-graphic
/// related content.  Calling base.Initialize will enumerate through any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
    base.Initialize();

    IsTrial = Guide.IsTrialMode;

    #region Pivot pages initialization
    this[0].Header = "Mercury";
    this[0].DrawScene += Mercury_DrawScene;
    this[0].DrawFrame += Mercury_DrawFrame;

    AddItem(new PivotGameItem());
    this[1].Header = "Venus";
    this[1].DrawScene += Venus_DrawScene;
    this[1].DrawFrame += Venus_DrawFrame;

    AddItem(new PivotGameItem());
    this[2].Header = "Earth";
    this[2].DrawScene += Earth_DrawScene;
    this[2].DrawFrame += Earth_DrawFrame;

    AddItem(new PivotGameItem());
    this[3].Header = "Mars";
    this[3].DrawScene += Mars_DrawScene;
    this[3].DrawFrame += Mars_DrawFrame;

    if (!IsTrial)
    {
        AddItem(new PivotGameItem());
        this[4].Header = "Jupiter";
        this[4].DrawScene += Jupiter_DrawScene;
        this[4].DrawFrame += Jupiter_DrawFrame;

        AddItem(new PivotGameItem());
        this[5].Header = "Saturn";
        this[5].DrawScene += Saturn_DrawScene;
        this[5].DrawFrame += Saturn_DrawFrame;

        AddItem(new PivotGameItem());
        this[6].Header = "Uranus";
        this[6].DrawScene += Uranus_DrawScene;
        this[6].DrawFrame += Uranus_DrawFrame;
```

```
                AddItem(new PivotGameItem());
                this[7].Header = "Neptune";
                this[7].DrawScene += Neptune_DrawScene;
                this[7].DrawFrame += Neptune_DrawFrame;
            }
            #endregion

            #region Pivot initialization
            SelectedIndex = 2;
            SelectionChanged += Planets_SelectionChanged;

            BackgroundColor = new Color(0,0,20);
            ForegroundColor = Color.SkyBlue;
            SceneBackgroundColor = new Color(0, 0, 10);
            SceneBackgroundTexture = Content.Load<Texture2D>("starfield");

            DrawingArea = new Rectangle(-480, 120, 1440, 480);

            SceneCenterTranslation = new Vector2(0, -0.5f);
            CameraScale = 8f;
            #endregion

        }

        //Is called when user moves to another tabs; Here the shapes are being updated
        void Planets_SelectionChanged(object sender, EventArgs e)
        {
            switch (SelectedIndex)
            {
                case 0: //Mercury
                    Planet = new Sphere(Vector3.Zero, 1, PRECISEMENT, tMercury, GraphicsDevice);
                    Reset(); break;
                case 1: //Venus
                    Planet = new Sphere(Vector3.Zero, 1.2f, PRECISEMENT, tVenus, GraphicsDevice);
                    Reset(); break;
                case 2: //Earth
                    Earth_Init();
                    Reset(); break;
                case 3: //Mars
                    Planet = new Sphere(Vector3.Zero, 1.3f, PRECISEMENT, tMars, GraphicsDevice);
                    Planet.Rotate(-Vector3.UnitZ, MARS_OBLIQUITY);
                    Moon = new Ellipsoid(new Vector3(0, 0, 2.5f), 0.1f, 1.2f, 0.83f, 1, PRECISEMENT/3+1,
tPhobos, GraphicsDevice); //Phobos
                    Moon.RotateComposition(-Vector3.UnitZ, MARS_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble()*MathHelper.Pi);
                    Moon2 = new Ellipsoid(new Vector3(0, 0, 4), 0.07f, 1.22f, 1, 0.85f, PRECISEMENT/3+1,
tDeimos, GraphicsDevice); //Deimos
                    Moon2.RotateComposition(-Vector3.UnitZ, MARS_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Reset(); break;
                case 4: //Jupiter
                    Planet = new Ellipsoid(Vector3.Zero, 2f, 1.06f, 1, 1.06f, PRECISEMENT, tJupiter,
GraphicsDevice);
                    Moon = new Sphere(new Vector3(0, 0, 3.5f), 0.1f, PRECISEMENT/3+1, tEuropa,
GraphicsDevice); //Europa
                    Moon.Rotate(Planet.Axis, -(float)randomAngle.NextDouble()*MathHelper.Pi);
                    Moon2 = new Sphere(new Vector3(0, 0, 3f), 0.11f, PRECISEMENT/3+1, tIo,
GraphicsDevice); //Io
                    Moon2.Rotate(Planet.Axis, -(float)randomAngle.NextDouble()*MathHelper.Pi);
                    Moon3 = new Sphere(new Vector3(0, 0, 4.3f), 0.17f, PRECISEMENT/3+2, tGanymede,
GraphicsDevice); //Ganymede
                    Moon3.Rotate(Planet.Axis, -(float)randomAngle.NextDouble()*MathHelper.Pi);
                    Moon4 = new Sphere(new Vector3(0, 0, 5.5f), 0.14f, PRECISEMENT/3+2, tCallisto,
GraphicsDevice); //Callisto
                    Moon4.Rotate(Planet.Axis, -(float)randomAngle.NextDouble()*MathHelper.Pi);
                    Reset(); break;
                case 5: //Saturn
                    Planet = new Ellipsoid(Vector3.Zero, 1.8f, 1.1f, 1, 1.1f, PRECISEMENT, tSaturn,
GraphicsDevice);
                    Planet.Rotate(-Vector3.UnitZ, SATURN_OBLIQUITY);
                    for (int i = 0; i < SaturnRing.Length; i++)
                    {
                        SaturnRing[i] = new FlatRingSector(Planet.Center, 2.7f, 0.9f, 1, MathHelper.TwoPi
/ SaturnRing.Length, tSaturnRing, GraphicsDevice);
                        SaturnRing[i].RotateComposition(-Vector3.UnitZ, SATURN_OBLIQUITY, Planet.Axis,
MathHelper.TwoPi / SaturnRing.Length * i);
```

```
                    }
                    Moon = new Sphere(new Vector3(0, 0, 4f), 0.15f, PRECISEMENT/3+3, tTitan,
GraphicsDevice); //Titan
                    Moon.RotateComposition(-Vector3.UnitZ, SATURN_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon2 = new Sphere(new Vector3(0, 0, 3f), 0.03f, PRECISEMENT/6, tRhea,
GraphicsDevice); //Rhea
                    Moon2.RotateComposition(-Vector3.UnitZ, SATURN_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon3 = new Sphere(new Vector3(0, 0, 2.6f), 0.027f, PRECISEMENT/6, tDione,
GraphicsDevice); //Dione
                    Moon3.RotateComposition(-Vector3.UnitZ, SATURN_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon4 = new Sphere(new Vector3(0, 0, 2.45f), 0.023f, PRECISEMENT/6, tTethys,
GraphicsDevice); //Tethys
                    Moon4.RotateComposition(-Vector3.UnitZ, SATURN_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon5 = new Sphere(new Vector3(0, 0, 7f), 0.06f, PRECISEMENT/3+3, tIapetus,
GraphicsDevice); //Iapetus
                    MoonAxis = (Vector3)My.Quaternion.Rotate(Planet.Axis, Vector3.UnitZ, SATURN_OBLIQUITY
- IAPETUS_SATURN_OBLIQUITY);
                    Moon5.RotateComposition(-Vector3.UnitZ, IAPETUS_SATURN_OBLIQUITY, MoonAxis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon6 = new Sphere(new Vector3(0, 0, 2.3f), 0.015f, PRECISEMENT/6, tEnceladus,
GraphicsDevice); //Enceladus
                    Moon6.RotateComposition(-Vector3.UnitZ, SATURN_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon7 = new Sphere(new Vector3(0, 0, 2.1f), 0.01f, PRECISEMENT/6, tMimas,
GraphicsDevice); //Mimas
                    Moon7.RotateComposition(-Vector3.UnitZ, SATURN_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);

                    //To see rings at once:
                    Planet.Rotate(Vector3.UnitX, 0.1f);Moon.Rotate(Vector3.UnitX,
0.1f);Moon2.Rotate(Vector3.UnitX, 0.1f);
                    Moon3.Rotate(Vector3.UnitX, 0.1f);Moon4.Rotate(Vector3.UnitX,
0.1f);Moon5.Rotate(Vector3.UnitX, 0.1f);
                    Moon6.Rotate(Vector3.UnitX, 0.1f);Moon7.Rotate(Vector3.UnitX, 0.1f);
                    foreach (FlatRingSector f in SaturnRing) f.Rotate(Vector3.UnitX, 0.1f);
                    Reset(); break;
                case 6: //Uranus
                    Planet = new Sphere(Vector3.Zero, 1.7f, PRECISEMENT, tUranus, GraphicsDevice);
                    Planet.Rotate(-Vector3.UnitZ, URANUS_OBLIQUITY);

                    Moon = new Sphere(new Vector3(0, 0, 3.5f), 0.11f, PRECISEMENT/3+1, tTitania,
GraphicsDevice); //Titania
                    Moon.RotateComposition(-Vector3.UnitZ, URANUS_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon2 = new Sphere(new Vector3(0, 0, 3.8f), 0.11f, PRECISEMENT/3+1, tOberon,
GraphicsDevice); //Oberon
                    Moon2.RotateComposition(-Vector3.UnitZ, URANUS_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon3 = new Sphere(new Vector3(0, 0, 2.6f), 0.09f, PRECISEMENT/3+1, tAriel,
GraphicsDevice); //Ariel
                    Moon3.RotateComposition(-Vector3.UnitZ, URANUS_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon4 = new Sphere(new Vector3(0, 0, 3f), 0.09f, PRECISEMENT/3+1, tUmbriel,
GraphicsDevice); //Umbriel
                    Moon4.RotateComposition(-Vector3.UnitZ, URANUS_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon5 = new Sphere(new Vector3(0, 0, 2.4f), 0.05f, PRECISEMENT/3+1, tMiranda,
GraphicsDevice); //Miranda
                    Moon5.RotateComposition(-Vector3.UnitZ, URANUS_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Reset(); break;
                case 7: //Neptune
                    Planet = new Sphere(Vector3.Zero, 1.6f, PRECISEMENT, tNeptune, GraphicsDevice);
                    Planet.Rotate(-Vector3.UnitZ, NEPTUNE_OBLIQUITY);

                    Moon = new Sphere(new Vector3(0, 0, 2.6f), 0.14f, PRECISEMENT/2-1, tTriton,
GraphicsDevice); //Triton
                    MoonAxis = (Vector3)My.Quaternion.Rotate(Planet.Axis, Vector3.UnitZ, SATURN_OBLIQUITY
- TRITON_NEPTUNE_OBLIQUITY);
                    Moon.RotateComposition(-Vector3.UnitZ, TRITON_NEPTUNE_OBLIQUITY, MoonAxis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
```

```
                    Moon2 = new Ellipsoid(new Vector3(0, 0, 5f), 0.04f, 1, 1.1f, 1, PRECISEMENT/4,
tNereid, GraphicsDevice); //Nereid
                    Moon2.RotateComposition(-Vector3.UnitZ, NEPTUNE_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
                    Moon3 = new Sphere(new Vector3(0, 0, 2f), 0.05f, PRECISEMENT/4, tProteus,
GraphicsDevice); //Proteus
                    Moon3.RotateComposition(-Vector3.UnitZ, NEPTUNE_OBLIQUITY, Planet.Axis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);

                    Reset(); break;
                }
            }

            //Resetting rotation angles and all that
            void Reset()
            {
                RotationAngle = 0;
                Zoom = 1;
                qAxis = DCL.Maths.Quaternion.j;
                if(SelectedIndex==2 && Latitude<=180 && UseGPS)
                    UpdatePositionVisualisation();
            }

            /// <summary>
            /// LoadContent will be called once per game and is the place to load
            /// all of your content.
            /// </summary>
            protected override void LoadContent()
            {
                SplashScreenImage = Content.Load<Texture2D>("SplashScreenImage");

                base.LoadContent();

                #region Fonts
                //Defining our fonts (base.LoadContent() allows to use the standard ones)
                HeaderFont = Content.Load<SpriteFont>("Pericles44");
                TitleFont = Content.Load<SpriteFont>("Pericles16");
                ContentFont = Content.Load<SpriteFont>("Segoe16");

                tinyFont = Content.Load<SpriteFont>("Segoe8");
                smallFont = Content.Load<SpriteFont>("Segoe12");
                #endregion

                #region Text
                Title = "Planets";
                LoadTextContent();
                #endregion

                #region Textures
                //Loading all the textures at once (no delays during the run time)
                tMercury = Content.Load<Texture2D>("mercury");
                tVenus = Content.Load<Texture2D>("venus");
                tEarth = Content.Load<Texture2D>("earth");
                    tMoon = Content.Load<Texture2D>("moon");
                tMars = Content.Load<Texture2D>("mars");
                    tPhobos = Content.Load<Texture2D>("phobos");
                    tDeimos = Content.Load<Texture2D>("deimos");
                tJupiter = Content.Load<Texture2D>("jupiter");
                    tEuropa = Content.Load<Texture2D>("europa");
                    tIo = Content.Load<Texture2D>("io");
                    tGanymede = Content.Load<Texture2D>("ganymede");
                    tCallisto = Content.Load<Texture2D>("callisto");
                tSaturn = Content.Load<Texture2D>("saturn");
                    tSaturnRing = Content.Load<Texture2D>("sat_ring_color");
                    tTitan = Content.Load<Texture2D>("titan");
                    tIapetus = Content.Load<Texture2D>("iapetus");
                    tEnceladus = Content.Load<Texture2D>("enceladus");
                    tMimas = Content.Load<Texture2D>("mimas");
                    tTethys = Content.Load<Texture2D>("tethys");
                    tRhea = Content.Load<Texture2D>("rhea");
                    tDione = Content.Load<Texture2D>("dione");
                tUranus = Content.Load<Texture2D>("uranus");
                    tTitania = Content.Load<Texture2D>("titania");
                    tOberon = Content.Load<Texture2D>("oberon");
                    tAriel = Content.Load<Texture2D>("ariel");
                    tUmbriel = Content.Load<Texture2D>("umbriel");
```

```
            tMiranda = Content.Load<Texture2D>("miranda");
         tNeptune = Content.Load<Texture2D>("neptune");
            tTriton = Content.Load<Texture2D>("triton");
            tNereid = Content.Load<Texture2D>("nereid");
            tProteus = Content.Load<Texture2D>("proteus");

        LoadIcons();
        #endregion

        Earth_Init();
    }

    /// <summary>
    /// UnloadContent will be called once per game and is the place to unload
    /// all content.
    /// </summary>
    /*protected override void UnloadContent()
    {

    }*/

    /// <summary>
    /// Allows the game to run logic such as updating the world,
    /// checking for collisions, gathering input, and playing audio.
    /// </summary>
    /// <param name="gameTime">Provides a snapshot of timing values.</param>
    protected override void Update(GameTime gameTime)
    {
        #region Handling touches
        Touches = TouchPanel.GetState();
        if (Touches.Count == 1)
        {
            #region One touch => Rotate
            if (Touches[0].State == TouchLocationState.Moved &&
                   Touches[0].Position.X > this[SelectedIndex].DrawingArea.X &&
                   Touches[0].Position.X < this[SelectedIndex].DrawingArea.X +
this[SelectedIndex].DrawingArea.Width &&
                   Touches[0].Position.Y > this[SelectedIndex].DrawingArea.Y &&
                   Touches[0].Position.Y < this[SelectedIndex].DrawingArea.Y +
this[SelectedIndex].DrawingArea.Height)
            {
                TouchLocation prevTouch;
                if (Touches[0].TryGetPreviousLocation(out prevTouch))
                {
                    float deltaX = Touches[0].Position.X - prevTouch.Position.X;
                    float deltaY = Touches[0].Position.Y - prevTouch.Position.Y;

                    if (Math.Abs(deltaX) > 0 || Math.Abs(deltaY) > 0)
                    {

                        qNew = (new My.Quaternion(0, deltaY, deltaX,
0)).Normalize().Approximate(0.5f);
                        qAxis = qNew;
                    }
                    RotationAngle = (qAxis.Y * deltaX + qAxis.X * deltaY) / 100;
                }
            }
            else
                RotationAngle = 0;
            #endregion

            #region One touch => Click on a Button

            if (IsTrial && Touches[0].State == TouchLocationState.Pressed &&
                   Touches[0].Position.X > 300 && Touches[0].Position.X < 430 &&
                   Touches[0].Position.Y > 0 && Touches[0].Position.Y < 40)
                Guide.ShowMarketplace(PlayerIndex.One);

            if (Touches[0].State == TouchLocationState.Pressed && SelectedIndex==2 &&
                   Touches[0].Position.X > 435 && Touches[0].Position.X < 480 &&
                   Touches[0].Position.Y > 0 && Touches[0].Position.Y < 40)
                if (IsTrial)
                    Guide.BeginShowMessageBox("Trial version", "Location services are available only
in the full version!", new string[] { "Buy now", "Buy later" }, 0, MessageBoxIcon.Alert, new
AsyncCallback(OnBuyMessageClosed), null);
                else
```

```
                    Guide.BeginShowMessageBox("Location settings", "This application can use the
built-in location services.\n\nYour location will be used ONLY to indicate your position on the
globe.\n\nEnable the access to and use of location from the location services?", new string[] { "Enable",
"Disable" }, 0, MessageBoxIcon.Alert, new AsyncCallback(OnGPSSettingsClosed), null);
                #endregion
            }
            else if (Touches.Count > 1 && !ChangingPages)
            {
                #region Multitouch => Resize
                RotationAngle = 0;

                //first touch
                if (Touches[0].State == TouchLocationState.Pressed || Touches[1].State ==
TouchLocationState.Pressed)
                {
                    touchesDistance = Math.Sqrt((Touches[0].Position.X - Touches[1].Position.X) *
(Touches[0].Position.X - Touches[1].Position.X) +
                                                (Touches[0].Position.Y - Touches[1].Position.Y) *
(Touches[0].Position.Y - Touches[1].Position.Y));
                }
                //moving fingers
                else if (Touches[0].State == TouchLocationState.Moved || Touches[1].State ==
TouchLocationState.Moved)
                {
                    double newTouchesDistance = Math.Sqrt((Touches[0].Position.X - Touches[1].Position.X)
* (Touches[0].Position.X - Touches[1].Position.X) +
                                                (Touches[0].Position.Y - Touches[1].Position.Y) *
(Touches[0].Position.Y - Touches[1].Position.Y));
                    Zoom = (float)(newTouchesDistance / touchesDistance) * Zoom;
                    Zoom = Math.Max(Math.Min(3f/Planet.Radius, Zoom), 0.7f);
                    touchesDistance = newTouchesDistance;
                }
                #endregion
            }
            #endregion

            #region Rotation

            switch (SelectedIndex)
            {
                case 0: //Mercury
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, MERCURY_ROTATION_SPEED);
                    break;
                case 1: //Venus
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, VENUS_ROTATION_SPEED);
                    break;
                case 2: //Earth
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, EARTH_ROTATION_SPEED);

                    MoonAxis = (Vector3)My.Quaternion.Rotate(MoonAxis, qAxis, RotationAngle);
                    Moon.RotateComposition(qAxis, RotationAngle, MoonAxis, MOON_ROTATION_SPEED);
                    break;
                case 3: //Mars
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, MARS_ROTATION_SPEED);

                    Moon.RotateComposition(qAxis, RotationAngle, Planet.Axis, PHOBOS_ROTATION_SPEED);
                    Moon2.RotateComposition(qAxis, RotationAngle, Planet.Axis, DEIMOS_ROTATION_SPEED);
                    break;
                case 4: //Jupiter
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, JUPITER_ROTATION_SPEED);

                    Moon.RotateComposition(qAxis, RotationAngle, Planet.Axis, EUROPA_ROTATION_SPEED);
                    Moon2.RotateComposition(qAxis, RotationAngle, Planet.Axis, IO_ROTATION_SPEED);
                    Moon3.RotateComposition(qAxis, RotationAngle, Planet.Axis, GANYMEDE_ROTATION_SPEED);
                    Moon4.RotateComposition(qAxis, RotationAngle, Planet.Axis, CALLISTO_ROTATION_SPEED);
                    break;
                case 5: //Saturn
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, SATURN_ROTATION_SPEED);

                    Moon.RotateComposition(qAxis, RotationAngle, Planet.Axis, TITAN_ROTATION_SPEED);
                    Moon2.RotateComposition(qAxis, RotationAngle, Planet.Axis, RHEA_ROTATION_SPEED);
                    Moon3.RotateComposition(qAxis, RotationAngle, Planet.Axis, DIONE_ROTATION_SPEED);
                    Moon4.RotateComposition(qAxis, RotationAngle, Planet.Axis, TETHYS_ROTATION_SPEED);
                    MoonAxis = (Vector3)My.Quaternion.Rotate(MoonAxis, qAxis, RotationAngle);
                    Moon5.RotateComposition(qAxis, RotationAngle, MoonAxis, IAPETUS_ROTATION_SPEED);
//Iapetus has another axis
```

```
                    Moon6.RotateComposition(qAxis, RotationAngle, Planet.Axis, ENCELADUS_ROTATION_SPEED);
                    Moon7.RotateComposition(qAxis, RotationAngle, Planet.Axis, MIMAS_ROTATION_SPEED);

                    foreach (FlatRingSector f in SaturnRing)
                        f.Rotate(qAxis, RotationAngle);
                    break;
                case 6: //Uran
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, URANUS_ROTATION_SPEED);

                    Moon.RotateComposition(qAxis, RotationAngle, Planet.Axis, TITANIA_ROTATION_SPEED);
                    Moon2.RotateComposition(qAxis, RotationAngle, Planet.Axis, OBERON_ROTATION_SPEED);
                    Moon3.RotateComposition(qAxis, RotationAngle, Planet.Axis, ARIEL_ROTATION_SPEED);
                    Moon4.RotateComposition(qAxis, RotationAngle, Planet.Axis, UMBRIEL_ROTATION_SPEED);
                    Moon5.RotateComposition(qAxis, RotationAngle, Planet.Axis, MIRANDA_ROTATION_SPEED);
                    break;
                case 7: //Neptun
                    Planet.RotateComposition(qAxis, RotationAngle, Planet.Axis, EARTH_ROTATION_SPEED);

                    MoonAxis = (Vector3)My.Quaternion.Rotate(MoonAxis, qAxis, RotationAngle);
                    Moon.RotateComposition(qAxis, RotationAngle, MoonAxis, TRITON_ROTATION_SPEED);
                    Moon2.RotateComposition(qAxis, RotationAngle, Planet.Axis, NEREID_ROTATION_SPEED);
                    Moon3.RotateComposition(qAxis, RotationAngle, Planet.Axis, PROTEUS_ROTATION_SPEED);
                    break;
            }
            #endregion

            #region Updating texts
            //New Axis
            //tAxis = String.Format("Axis: ({0:G4}; {1:G4}; {2:G4})", qAxis.X, qAxis.Y, qAxis.Z);

            //ANGLES OF ROTATION
            //angAroundAxis += MathHelper.ToDegrees(0.025f); if (angAroundAxis > 360) angAroundAxis -=
360;
            //angAroundEarth += MathHelper.ToDegrees(0.005f); if (angAroundEarth > 360) angAroundEarth -=
360;

            //tAngleA = String.Format("Around axis:  {0}°", (int)angAroundAxis);
            //tAngleB = String.Format("Around earth: {0}°", (int)angAroundEarth);
            #endregion

            base.Update(gameTime);
        }

        #region Drawing code for each tab
        #region Mercury
        private void Mercury_DrawScene(object sender, EventArgs e)
        {
            Planet.Draw(BasicEffect, DrawMode.Textured);
        }
        private void Mercury_DrawFrame(object sender, EventArgs e)
        {
            SpriteBatch.Begin();
            DrawString(ContentFont, mercuryDiameter, plDiameter, Color.White);
            DrawString(ContentFont, mercuryMass, plMass, Color.White);
            DrawString(ContentFont, mercuryDistance, plDistance, Color.White);
            DrawString(ContentFont, mercuryRotationPeriod, plRotationPeriod, Color.White);
            DrawString(ContentFont, mercuryRevolutionPeriod, plRevolutionPeriod, Color.White);
            DrawString(ContentFont, mercuryTemperatures, plTemperatures, Color.White);
            DrawString(ContentFont, mercuryGravitationalPull, plGravitationalPull, Color.White);
            DrawString(TitleFont, mercuryMoons, plMoons, Color.White);
            SpriteBatch.End();
        }
        #endregion

        #region Venus
        private void Venus_DrawScene(object sender, EventArgs e)
        {
            Planet.Draw(BasicEffect, DrawMode.Textured);
        }
        private void Venus_DrawFrame(object sender, EventArgs e)
        {
            SpriteBatch.Begin();
            DrawString(ContentFont, venusDiameter, plDiameter, Color.White);
            DrawString(ContentFont, venusMass, plMass, Color.White);
            DrawString(ContentFont, venusDistance, plDistance, Color.White);
            DrawString(ContentFont, venusRotationPeriod, plRotationPeriod, Color.White);
```

```
    DrawString(ContentFont, venusRevolutionPeriod, plRevolutionPeriod, Color.White);
    DrawString(ContentFont, venusTemperatures, plTemperatures, Color.White);
    DrawString(ContentFont, venusGravitationalPull, plGravitationalPull, Color.White);
    DrawString(TitleFont, venusMoons, plMoons, Color.White);
    SpriteBatch.End();
}
#endregion

#region Earth
private void Earth_DrawScene(object sender, EventArgs e)
{
    //There are only earth, moon and dot and we need them to be fast,
    //so we don't use Shapes.DrawScene.
    if (Moon.Center.Z > 0)
    {
        Planet.Draw(BasicEffect, DrawMode.Textured);
        Moon.Draw(BasicEffect, DrawMode.Textured);
    }
    else
    {
        Moon.Draw(BasicEffect, DrawMode.Textured);
        Planet.Draw(BasicEffect, DrawMode.Textured);
    }
}
private void Earth_DrawFrame(object sender, EventArgs e)
{
    SpriteBatch.Begin();

    if (!IsTrial)
    {
        SpriteBatch.DrawString(smallFont, tLocation1, coordPos1, Color.SkyBlue);
        SpriteBatch.DrawString(smallFont, tLocation2, coordPos2, Color.SkyBlue);
    }

    switch(gpsState)
    {
        default: SpriteBatch.Draw(icoGPSoff, coordGps, Color.White); break;
        case 1: SpriteBatch.Draw(icoGPSsearch, coordGps, Color.White); break;
        case 2: SpriteBatch.Draw(icoGPSon, coordGps, Color.White); break;
    }

    DrawString(ContentFont, earthDiameter, plDiameter, Color.White);
    DrawString(ContentFont, earthMass, plMass, Color.White);
    DrawString(ContentFont, earthDistance, plDistance, Color.White);
    DrawString(ContentFont, earthRotationPeriod, plRotationPeriod, Color.White);
    DrawString(ContentFont, earthRevolutionPeriod, plRevolutionPeriod, Color.White);
    DrawString(ContentFont, earthTemperatures, plTemperatures, Color.White);
    DrawString(ContentFont, earthGravitationalPull, plGravitationalPull, Color.White);
    DrawString(TitleFont, earthMoons, plMoons, Color.White);

    DrawSprite(icoMoon, posMoon, Color.White);
    DrawString(ContentFont, strMoon, coordMoon, Color.White);
    SpriteBatch.End();
}
#endregion

#region Mars
private void Mars_DrawScene(object sender, EventArgs e)
{
    Shape.DrawScene(BasicEffect, DrawMode.Textured, Planet, Moon, Moon2);
}
private void Mars_DrawFrame(object sender, EventArgs e)
{
    SpriteBatch.Begin();
    DrawString(ContentFont, marsDiameter, plDiameter, Color.White);
    DrawString(ContentFont, marsMass, plMass, Color.White);
    DrawString(ContentFont, marsDistance, plDistance, Color.White);
    DrawString(ContentFont, marsRotationPeriod, plRotationPeriod, Color.White);
    DrawString(ContentFont, marsRevolutionPeriod, plRevolutionPeriod, Color.White);
    DrawString(ContentFont, marsTemperatures, plTemperatures, Color.White);
    DrawString(ContentFont, marsGravitationalPull, plGravitationalPull, Color.White);
    DrawString(TitleFont, marsMoons, plMoons, Color.White);

    DrawSprite(icoPhobos, posPhobos, Color.White);
    DrawString(ContentFont, strPhobos, coordPhobos, Color.White);
    DrawSprite(icoDeimos, posDeimos, Color.White);
```

```
            DrawString(ContentFont, strDeimos, coordDeimos, Color.White);

            SpriteBatch.End();
        }
        #endregion

        #region Jupiter
        private void Jupiter_DrawScene(object sender, EventArgs e)
        {
            //Shape.DrawScene is preferable when we have many moons
            Shape.DrawScene(BasicEffect, DrawMode.Textured, Planet, Moon, Moon2, Moon3, Moon4);
        }
        private void Jupiter_DrawFrame(object sender, EventArgs e)
        {
            SpriteBatch.Begin();
            DrawString(ContentFont, jupiterDiameter, plDiameter, Color.White);
            DrawString(ContentFont, jupiterMass, plMass, Color.White);
            DrawString(ContentFont, jupiterDistance, plDistance, Color.White);
            DrawString(ContentFont, jupiterRotationPeriod, plRotationPeriod, Color.White);
            DrawString(ContentFont, jupiterRevolutionPeriod, plRevolutionPeriod, Color.White);
            DrawString(ContentFont, jupiterTemperatures, plTemperatures, Color.White);
            DrawString(ContentFont, jupiterGravitationalPull, plGravitationalPull, Color.White);
            DrawString(TitleFont, jupiterMoons, plMoons, Color.White);

            DrawSprite(icoIo, posIo, Color.White);
            DrawString(ContentFont, strIo, coordIo, Color.White);
            DrawSprite(icoEuropa, posEuropa, Color.White);
            DrawString(ContentFont, strEuropa, coordEuropa, Color.White);
            DrawSprite(icoGanymede, posGanymede, Color.White);
            DrawString(ContentFont, strGanymede, coordGanymede, Color.White);
            DrawSprite(icoCallisto, posCallisto, Color.White);
            DrawString(ContentFont, strCallisto, coordCallisto, Color.White);
            SpriteBatch.End();
        }
        #endregion

        #region Saturn
        private void Saturn_DrawScene(object sender, EventArgs e)
        {
            Shape.DrawScene(BasicEffect, DrawMode.Textured, SaturnRing, Planet, Moon, Moon2, Moon3, Moon4,
Moon5, Moon6, Moon7);
        }
        private void Saturn_DrawFrame(object sender, EventArgs e)
        {
            SpriteBatch.Begin();
            DrawString(ContentFont, saturnDiameter, plDiameter, Color.White);
            DrawString(ContentFont, saturnMass, plMass, Color.White);
            DrawString(ContentFont, saturnDistance, plDistance, Color.White);
            DrawString(ContentFont, saturnRotationPeriod, plRotationPeriod, Color.White);
            DrawString(ContentFont, saturnRevolutionPeriod, plRevolutionPeriod, Color.White);
            DrawString(ContentFont, saturnTemperatures, plTemperatures, Color.White);
            DrawString(ContentFont, saturnGravitationalPull, plGravitationalPull, Color.White);
            DrawString(TitleFont, saturnMoons, plMoons, Color.White);

            DrawSprite(icoMimas, posMimas, Color.White);
            DrawString(ContentFont, strMimas, coordMimas, Color.White);
            DrawSprite(icoEnceladus, posEnceladus, Color.White);
            DrawString(ContentFont, strEnceladus, coordEnceladus, Color.White);
            DrawSprite(icoTethys, posTethys, Color.White);
            DrawString(ContentFont, strTethys, coordTethys, Color.White);
            DrawSprite(icoDione, posDione, Color.White);
            DrawString(ContentFont, strDione, coordDione, Color.White);
            DrawSprite(icoRhea, posRhea, Color.White);
            DrawString(ContentFont, strRhea, coordRhea, Color.White);
            DrawSprite(icoTitan, posTitan, Color.White);
            DrawString(ContentFont, strTitan, coordTitan, Color.White);
            DrawSprite(icoIapetus, posIapetus, Color.White);
            DrawString(ContentFont, strIapetus, coordIapetus, Color.White);
            SpriteBatch.End();
        }
        #endregion

        #region Uranus
        private void Uranus_DrawScene(object sender, EventArgs e)
        {
            Shape.DrawScene(BasicEffect, DrawMode.Textured, Planet, Moon, Moon2, Moon3, Moon4, Moon5);
```

```
}
private void Uranus_DrawFrame(object sender, EventArgs e)
{
    SpriteBatch.Begin();
    DrawString(ContentFont, uranusDiameter, plDiameter, Color.White);
    DrawString(ContentFont, uranusMass, plMass, Color.White);
    DrawString(ContentFont, uranusDistance, plDistance, Color.White);
    DrawString(ContentFont, uranusRotationPeriod, plRotationPeriod, Color.White);
    DrawString(ContentFont, uranusRevolutionPeriod, plRevolutionPeriod, Color.White);
    DrawString(ContentFont, uranusTemperatures, plTemperatures, Color.White);
    DrawString(ContentFont, uranusGravitationalPull, plGravitationalPull, Color.White);
    DrawString(TitleFont, uranusMoons, plMoons, Color.White);

    DrawSprite(icoMiranda, posMiranda, Color.White);
    DrawString(ContentFont, strMiranda, coordMiranda, Color.White);
    DrawSprite(icoAriel, posAriel, Color.White);
    DrawString(ContentFont, strAriel, coordAriel, Color.White);
    DrawSprite(icoUmbriel, posUmbriel, Color.White);
    DrawString(ContentFont, strUmbriel, coordUmbriel, Color.White);
    DrawSprite(icoTitania, posTitania, Color.White);
    DrawString(ContentFont, strTitania, coordTitania, Color.White);
    DrawSprite(icoOberon, posOberon, Color.White);
    DrawString(ContentFont, strOberon, coordOberon, Color.White);
    SpriteBatch.End();
}
#endregion

#region Neptune
private void Neptune_DrawScene(object sender, EventArgs e)
{
    Shape.DrawScene(BasicEffect, DrawMode.Textured, Planet, Moon, Moon2, Moon3);
}
private void Neptune_DrawFrame(object sender, EventArgs e)
{
    SpriteBatch.Begin();
    DrawString(ContentFont, neptuneDiameter, plDiameter, Color.White);
    DrawString(ContentFont, neptuneMass, plMass, Color.White);
    DrawString(ContentFont, neptuneDistance, plDistance, Color.White);
    DrawString(ContentFont, neptuneRotationPeriod, plRotationPeriod, Color.White);
    DrawString(ContentFont, neptuneRevolutionPeriod, plRevolutionPeriod, Color.White);
    DrawString(ContentFont, neptuneTemperatures, plTemperatures, Color.White);
    DrawString(ContentFont, neptuneGravitationalPull, plGravitationalPull, Color.White);
    DrawString(TitleFont, neptuneMoons, plMoons, Color.White);

    DrawSprite(icoProteus, posProteus, Color.White);
    DrawString(ContentFont, strProteus, coordProteus, Color.White);
    DrawSprite(icoTriton, posTriton, Color.White);
    DrawString(ContentFont, strTriton, coordTriton, Color.White);
    DrawSprite(icoNereid, posNereid, Color.White);
    DrawString(ContentFont, strNereid, coordNereid, Color.White);
    SpriteBatch.End();
}
#endregion
#endregion

protected override void Draw(GameTime gameTime)
{
    base.Draw(gameTime);

    SpriteBatch.Begin();

        DrawString(TitleFont, strCharacteristics, coordCharacteristics, Color.SkyBlue);
        DrawString(TitleFont, strMoons, coordMoons, Color.SkyBlue);
        DrawString(ContentFont, strDiameter, coordDiameter, Color.SkyBlue);
        DrawString(ContentFont, strMass, coordMass, Color.SkyBlue);
        DrawString(ContentFont, strDistance, coordDistance, Color.SkyBlue);
        DrawString(ContentFont, strRotationPeriod, coordRotationPeriod, Color.SkyBlue);
        DrawString(ContentFont, strRevolutionPeriod, coordRevolutionPeriod, Color.SkyBlue);
        DrawString(ContentFont, strTemperatures, coordTemperatures, Color.SkyBlue);
        DrawString(ContentFont, strGravitationalPull, coordGravitationalPull, Color.SkyBlue);
        DrawString(tinyFont, strRemark, coordNotation, Color.SkyBlue);

        DrawSprite(icoDiameter, spDiameter, Color.SkyBlue);
        DrawSprite(icoMass, spMass, Color.SkyBlue);
        DrawSprite(icoDistance, spDistance, Color.SkyBlue);
        DrawSprite(icoRotationPeriod, spRotationPeriod, Color.SkyBlue);
```

```
                DrawSprite(icoRevolutionPeriod, spRevolutionPeriod, Color.SkyBlue);
                DrawSprite(icoTemperatures, spTemperatures, Color.SkyBlue);
                DrawSprite(icoGravitationalPull, spGravitationalPull, Color.SkyBlue);

                if(IsTrial)
                    SpriteBatch.DrawString(TitleFont, strBuy, coordBuy, Color.White);

            SpriteBatch.End();
        }

        //Is called from two places in code
        void Earth_Init()
        {
            Planet = new Sphere(Vector3.Zero, 1.5f, PRECISEMENT, tEarth, GraphicsDevice);
            Planet.Rotate(-Vector3.UnitZ, EARTH_OBLIQUITY);
            Moon = new Sphere(new Vector3(3.5f, 0, 0), 0.25f, PRECISEMENT / 2+2, tMoon, GraphicsDevice);
            MoonAxis = (Vector3)My.Quaternion.Rotate(Planet.Axis, Vector3.UnitZ, EARTH_OBLIQUITY -
MOON_EARTH_OBLIQUITY);
            Moon.RotateComposition(-Vector3.UnitZ, MOON_EARTH_OBLIQUITY, MoonAxis, -
(float)randomAngle.NextDouble() * MathHelper.Pi);
        }
    }
}
```