

Game Engine Development II

Week1

Hooman Salamat




Instructor

Hooman Salamat (Lectures & Labs)

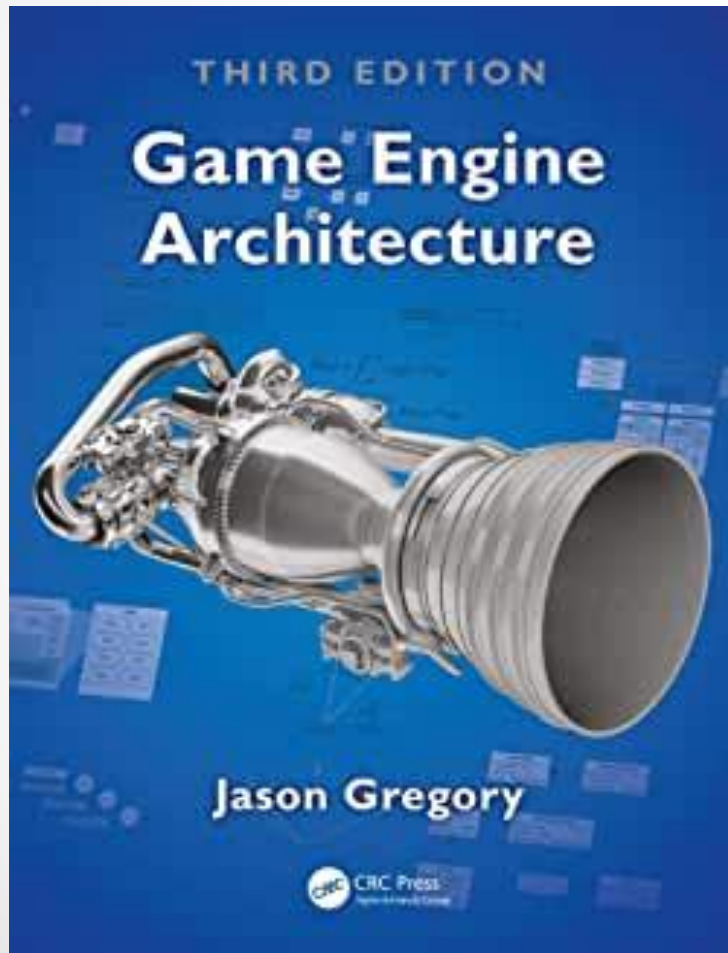
- Hooman.Salamat@georgebrown.ca
- Discord: Hooman#2526



Assessment

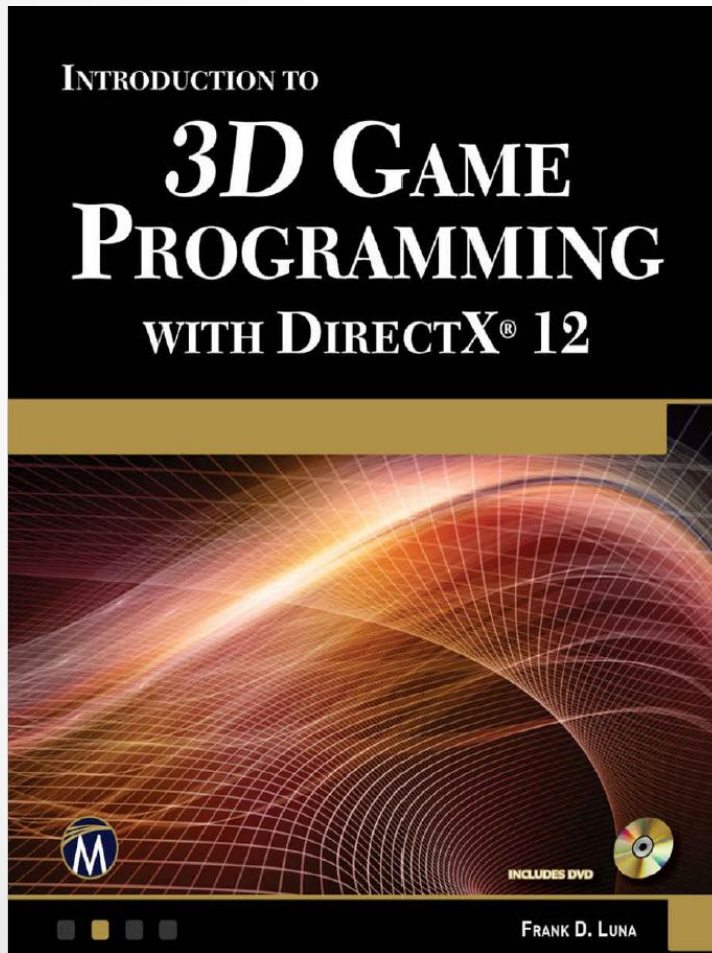
- 2x Assignments 20%
- 
- 1x Final Exam 30%
- 
- 1x Final Project 50%
- 

Textbook 1



- Game Engine Architecture, Third Edition
- By: Jason Gregory
- ISBN-13: 978-1-1380-3545-4
- Publisher: CRC Press

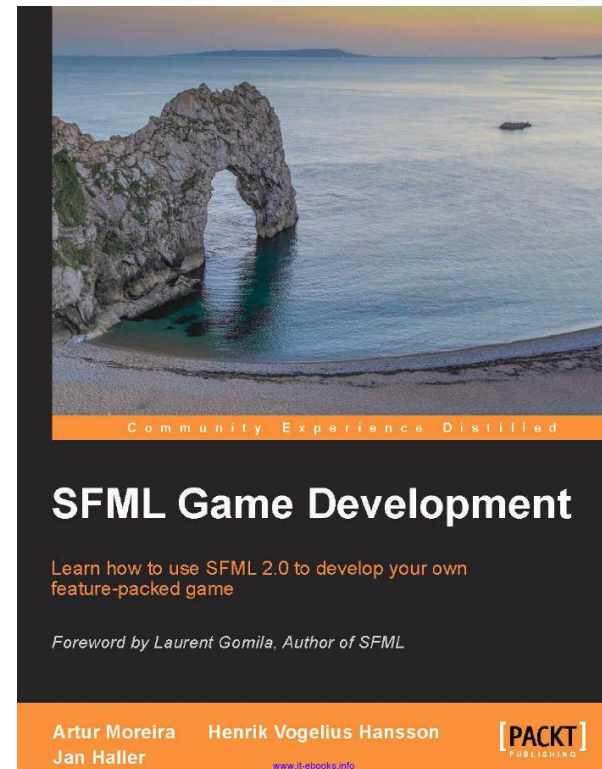
Textbook2



- Introduction to 3D Game Programming With DirectX12
- By: Frank D. Luna
- ISBN: 9781942270065
- Publisher: Mercury

Textbook3

- SFML Game Development
- Google it
- <https://www.packtpub.com/game-development/sfml-game-development>



Course Materials

- <https://github.com/hsalamat/GameEngineDevelopment2>
- Our repository
 - <https://github.com/hsalamat/GameEngineDevelopment2.git>
 - Open the command line (cmd)
 - Cd "to a location that you want to clone"
 - Git clone <https://github.com/hsalamat/GameEngineDevelopment2.git>
 - Cd GameEngineDevelopment2
 - Git status

Quaternions

Objectives:

1. To review the complex numbers and recall how complex number multiplication performs a rotation in the plane.
2. To obtain an understanding of quaternions and the operations defined on them.
3. To discover how the set of unit quaternions represent 3D rotations.
4. To find out how to convert between the various rotation representations.
5. To learn how to interpolate between unit quaternions, and understand that this is geometrically equivalent to interpolating between 3D orientations.
6. To become familiar with the DirectX Math library's quaternion functions and classes.



REVIEW OF THE COMPLEX NUMBERS

An ordered pair of real numbers $\mathbf{z} = (a, b)$ is a complex number.

The first component is called the *real* part and the second component is called the *imaginary* part.

$$1. (a, b) = (c, d) \text{ if and only if } a = c \text{ and } b = d.$$

$$2. (a, b) \pm (c, d) = (a \pm c, b \pm d).$$

$$3. (a, b)(c, d) = (ac - bd, ad + bc).$$

4. The *complex conjugate* of a complex number $\mathbf{z} = (a, b)$ is denoted by $\bar{\mathbf{z}} = (a, -b)$.

$$5. \frac{(a, b)}{(c, d)} = \left(\frac{ac+bd}{c^2+d^2}, \frac{bc-ad}{c^2+d^2} \right)$$

A simple way to remember the complex division formula is to multiply the numerator and denominator by the conjugate of the denominator so that the denominator

becomes a real number:

$$\frac{(a, b)}{(c, d)} = \frac{(a, b)(c, -d)}{(c, d)(c, -d)} = \left(\frac{ac + bd}{c^2 + d^2}, \frac{bc - ad}{c^2 + d^2} \right)$$

6. Any real number can be thought of as a complex number with a zero imaginary component: $x = (x, 0)$;

7. A real number times a complex number is given by $x(a, b) = (x, 0)(a, b) = (xa, xb) = (a, b)(x, 0) = (a, b)x$

8. We define the *imaginary unit* $i = (0, 1)$. Using our definition of complex multiplication, $i^2 = (0, 1)(0, 1) = (-1, 0) = -1$, which implies $i = \sqrt{-1}$

9. A complex number (a, b) can be written in the form $a + ib$. We have $a = (a, 0)$, $b = (b, 0)$ and $i = (0, 1)$, so:

$$a + ib = (a, 0) + (0, 1)(b, 0) = (a, 0) + (0, b) = (a, b)$$

$$10. a + ib \pm c + id = (a \pm c) + i(b \pm d)$$

$$11. (a + ib)(c + id) = (ac - bd) + i(ad + bc)$$

$$12. \frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i \frac{bc-ad}{c^2+d^2} \text{ if } (c, d) \neq (0, 0)$$

13. The complex conjugate of $\mathbf{z} = a + ib$ is given by $\bar{\mathbf{z}} = a - ib$

Geometric Interpretation

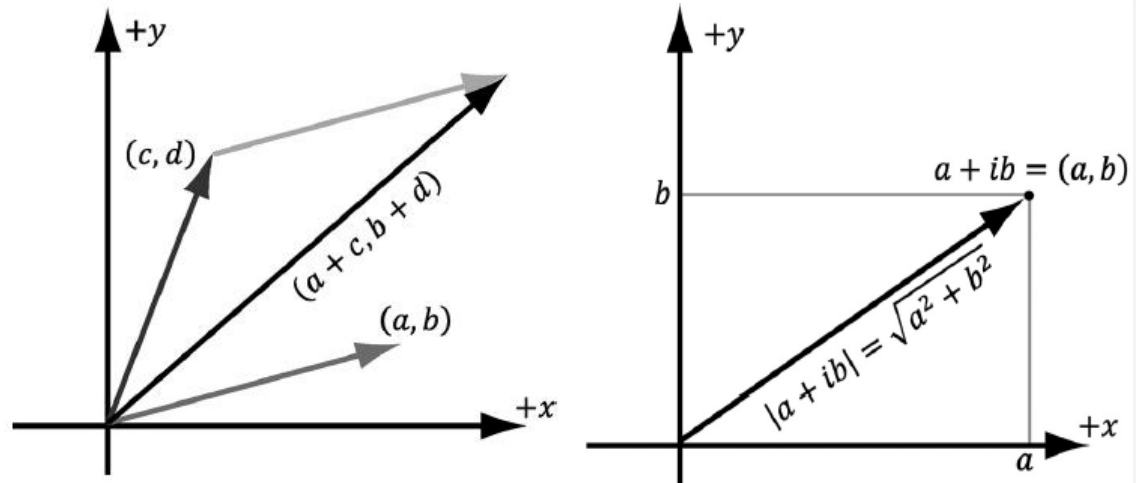
1. The ordered pair form $a + ib = (a, b)$ of a complex number naturally suggests that we think of a complex number geometrically as a 2D point or vector in the complex plane.

2. Complex addition is reminiscent of vector addition in the plane.

3. The *absolute value*, or *magnitude*, of the complex number $a + ib$ is defined as the length of the vector it represents which we know is given by:

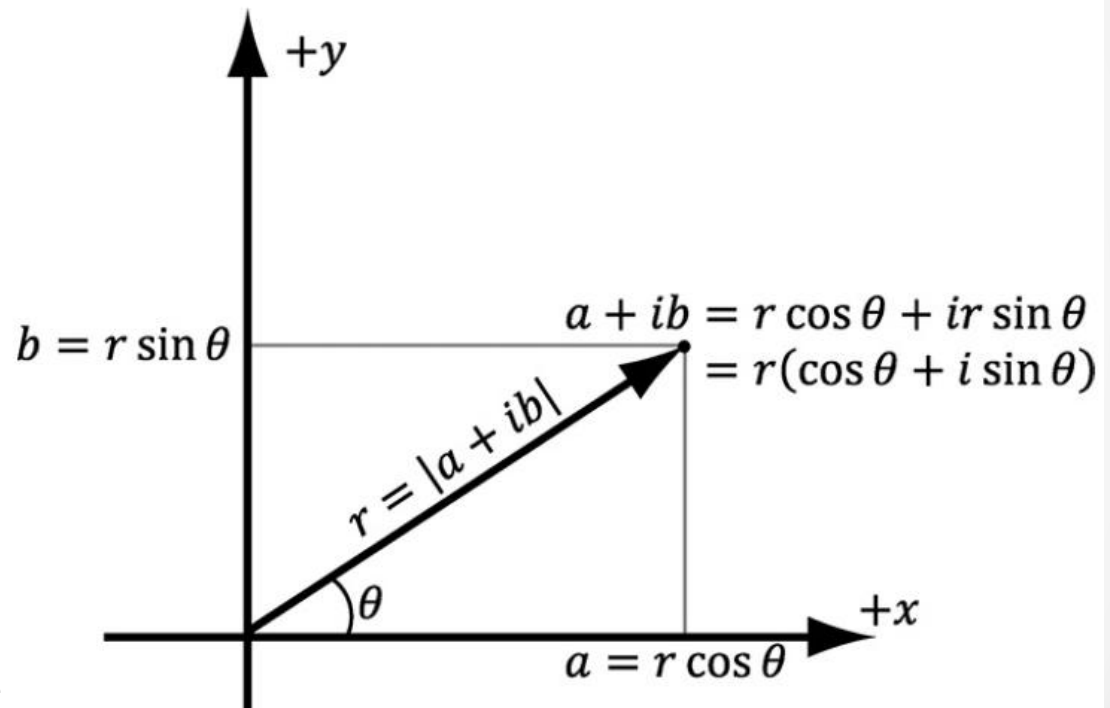
$$|a + ib| = \sqrt{a^2 + b^2}$$

4. A complex number is a *unit complex number* if it has a magnitude of one.



Polar Representation

Because complex numbers can be viewed as just points or vectors in the 2D complex plane, we can just as well express their components using polar coordinates



Rotations

Multiplying a complex number z_1 (thought of as a 2D vector or point) by a unit complex number z_2 results in a rotation of z_1 .

$$z_1 = r_1(\cos\theta_1 + i\sin\theta_1),$$

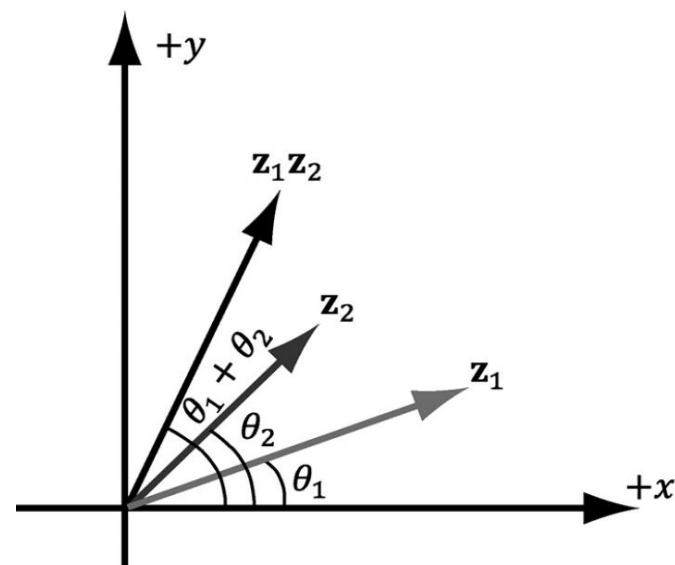
$$z_2 = r_2(\cos\theta_2 + i\sin\theta_2).$$

$$\sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$$

Recall $\cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$

The product $z_1 z_2$ rotates z_1 by the angle θ_2 .

$$\begin{aligned} z_1 z_2 &= r_1 r_2 (\cos\theta_1 \cos\theta_2 - \sin\theta_1 \sin\theta_2 + i(\cos\theta_1 \sin\theta_2 + \sin\theta_1 \cos\theta_2)) \\ &= r_1 r_2 (\cos(\theta_1 + \theta_2) + i\sin(\theta_1 + \theta_2)) \end{aligned}$$



QUATERNION ALGEBRA

An ordered 4-tuple of real numbers

$\mathbf{q} = (x, y, z, w) = (q_1, q_2, q_3, q_4)$ is a quaternion.

This is commonly abbreviated as

$\mathbf{q} = (\mathbf{u}, w) = (x, y, z, w)$, and we call $\mathbf{u} = (x, y, z)$ the

imaginary vector part and w the real part.

1. $(\mathbf{u}, a) = (\mathbf{v}, b)$ if and only if $\mathbf{u} = \mathbf{v}$ and $a = b$.

2. $(\mathbf{u}, a) \pm (\mathbf{v}, b) = (\mathbf{u} \pm \mathbf{v}, a \pm b)$.

3. $(\mathbf{u}, a)(\mathbf{v}, b) = (a\mathbf{v} + b\mathbf{u} + \mathbf{u} \times \mathbf{v}, ab - \mathbf{u} \cdot \mathbf{v})$

Let $\mathbf{p} = (\mathbf{u}, p_4) = (p_1, p_2, p_3, p_4)$ and $\mathbf{q} = (\mathbf{v}, q_4) = (q_1, q_2, q_3, q_4)$. Then $\mathbf{u} \times \mathbf{v} = (p_2q_3 -$

$p_3q_2, p_3q_1 - p_1q_3, p_1q_2 - p_2q_1)$ and $\mathbf{u} \cdot \mathbf{v} = p_1q_1 + p_2q_2 + p_3q_3$. Now, in component form, the

quaternion product $\mathbf{r} = \mathbf{pq}$ takes on the form:

$$r_1 = p_4q_1 + q_4p_1 + p_2q_3 - p_3q_2 = q_1p_4 - q_2p_3 + q_3p_2 + q_4p_1$$

$$r_2 = p_4q_2 + q_4p_2 + p_3q_1 - p_1q_3 = q_1p_3 + q_2p_4 - q_3p_1 + q_4p_2$$

$$r_3 = p_4q_3 + q_4p_3 + p_1q_2 - p_2q_1 = -q_1p_2 + q_2p_1 + q_3p_4 + q_4p_3$$

$$r_4 = p_4q_4 - p_1q_1 - p_2q_2 - p_3q_3 = -q_1p_1 - q_2p_2 - q_3p_3 + q_4p_4$$

This can be written as a matrix product:

$$\mathbf{pq} = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}$$

Special Products

Let $\mathbf{i} = (1, 0, 0, 0)$, $\mathbf{j} = (0, 1, 0, 0)$, $\mathbf{k} = (0, 0, 1, 0)$
be quaternions.

Then we have the special products, some of which are reminiscent of the behavior of the cross product:

$$\begin{aligned}\mathbf{i}^2 &= \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1 \\ \mathbf{ij} &= \mathbf{k} = -\mathbf{ji} \\ \mathbf{jk} &= \mathbf{i} = -\mathbf{kj} \\ \mathbf{ki} &= \mathbf{j} = -\mathbf{ik}\end{aligned}$$

Quaternion multiplication is *not* commutative that

$$\mathbf{ij} = -\mathbf{ji}.$$

Quaternion multiplication is associative, however; this can be seen from the fact that quaternion multiplication can be written using matrix multiplication and matrix multiplication is associative. The quaternion $\mathbf{e} = (0, 0, 0, 1)$ serves as a multiplicative identity:

$$\mathbf{pe} = \mathbf{ep} = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

We also have that quaternion multiplication distributes over quaternion addition:

$$\mathbf{p}(\mathbf{q} + \mathbf{r}) = \mathbf{pq} + \mathbf{pr} \text{ and } (\mathbf{q} + \mathbf{r})\mathbf{p} = \mathbf{qp} + \mathbf{rp}.$$

Conversions

We relate real numbers, vectors (or points), and quaternions in the following way:

Let s be a real number and let $\mathbf{u} = (x, y, z)$ be a vector. Then

1. $s = (0, 0, 0, s)$

2. $\mathbf{u} = (x, y, z) = (\mathbf{u}, 0) = (x, y, z, 0)$

In other words, any real number can be thought of as a quaternion with a zero vector part, and any vector can be thought of as a quaternion with zero real part.

The identity quaternion, $1 = (0, 0, 0, 1)$. A quaternion with zero real part is called a *pure quaternion*.

Observe, using the definition of quaternion multiplication, that a real number times a quaternion is just "scalar multiplication" and it is commutative:

$$s(p_1, p_2, p_3, p_4) = (0, 0, 0, s)(p_1, p_2, p_3, p_4) = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & s \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \begin{bmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{bmatrix}$$

Similarly,

$$(p_1, p_2, p_3, p_4)s = (p_1, p_2, p_3, p_4)(0, 0, 0, s) = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ s \end{bmatrix} = \begin{bmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{bmatrix}$$

Conjugate and Norm

The conjugate of a quaternion $\mathbf{q} = (q_1, q_2, q_3, q_4) = (\mathbf{u}, q_4)$ is denoted by \mathbf{q}^* and defined by

$$\mathbf{q}^* = -q_1 - q_2 - q_3 - q_4 = (-\mathbf{u}, q_4)$$

In other words, we just negate the imaginary vector part of the quaternion; compare this to the complex number conjugate. The conjugate has the following properties:

1. $(\mathbf{pq})^* = \mathbf{q}^* \mathbf{p}^*$
2. $(\mathbf{p} + \mathbf{q})^* = \mathbf{p}^* + \mathbf{q}^*$
3. $(\mathbf{q}^*)^* = \mathbf{q}$
4. $(s\mathbf{q})^* = s\mathbf{q}^*$ for $s \in \mathbb{R}$
5. $\mathbf{q} + \mathbf{q}^* = (\mathbf{u}, q_4) + (-\mathbf{u}, q_4) = (0, 2q_4) = 2q_4$
6. $\mathbf{qq}^* = \mathbf{q}^* \mathbf{q} = q_1^2 + q_2^2 + q_3^2 + q_4^2 = \|\mathbf{u}\|^2 + q_4^2$

$$\|\mathbf{q}\| = \sqrt{\mathbf{qq}^*} = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} = \sqrt{\|\mathbf{u}\|^2 + q_4^2}$$

The *norm* (or *magnitude*) of a quaternion is defined by:

We say that a quaternion is a *unit quaternion* if it has a norm of one. The norm has the

following properties: 1. $\|\mathbf{q}^*\| = \|\mathbf{q}\|$ 2. $\|\mathbf{pq}\| = \|\mathbf{p}\| \|\mathbf{q}\|$

In particular, property 2 tells us that the product of two unit quaternions is a unit

quaternion; also if $\|\mathbf{p}\| = 1$, then $\|\mathbf{pq}\| = \|\mathbf{q}\|$. The conjugate and norm properties can be derived straightforwardly from the definitions. For example,

$$(\mathbf{q}^*)^* = (-\mathbf{u}, q_4)^* = (\mathbf{u}, q_4) = \mathbf{q}$$

$$\|\mathbf{q}^*\| = \|(-\mathbf{u}, q_4)\| = \sqrt{\|-\mathbf{u}\|^2 + q_4^2} = \sqrt{\|\mathbf{u}\|^2 + q_4^2} = \|\mathbf{q}\|$$

$$\begin{aligned} \|\mathbf{pq}\|^2 &= (\mathbf{pq})(\mathbf{pq})^* \\ &= \mathbf{pq} \mathbf{q}^* \mathbf{p}^* \\ &= \mathbf{p} \|\mathbf{q}\|^2 \mathbf{p}^* \\ &= \mathbf{p} \mathbf{p}^* \|\mathbf{q}\|^2 \\ &= \|\mathbf{p}\|^2 \|\mathbf{q}\|^2 \end{aligned}$$

Inverses

As with matrices, quaternion multiplication is not commutative, so we cannot define a division operator.

However, every nonzero quaternion has an inverse. (The zero quaternion has zeros for all its components.) Let $\mathbf{q} = (q_1, q_2, q_3, q_4) = (\mathbf{u}, q_4)$ be a nonzero quaternion, then the inverse is denoted by \mathbf{q}^{-1} and given by:

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{\|\mathbf{q}\|^2}$$

$$(\mathbf{q}^{-1})^{-1} = \mathbf{q}$$

$$(\mathbf{pq})^{-1} = \mathbf{q}^{-1}\mathbf{p}^{-1}$$



Polar Representation

If $\mathbf{q} = (q_1, q_2, q_3, q_4) = (\mathbf{u}, q_4)$ is a unit quaternion, then $\|\mathbf{q}\|^2 = \|\mathbf{u}\|^2 + q_4^2 = 1$

Figure shows there exists an angle $\theta \in [0, \pi]$ such that $q_4 = \cos \theta$.

Employing the trigonometric identity

$\sin^2 \theta + \cos^2 \theta = 1$, we have that

$$\sin^2 \theta = 1 - \cos^2 \theta = 1 - q_4^2 = \|\mathbf{u}\|^2$$

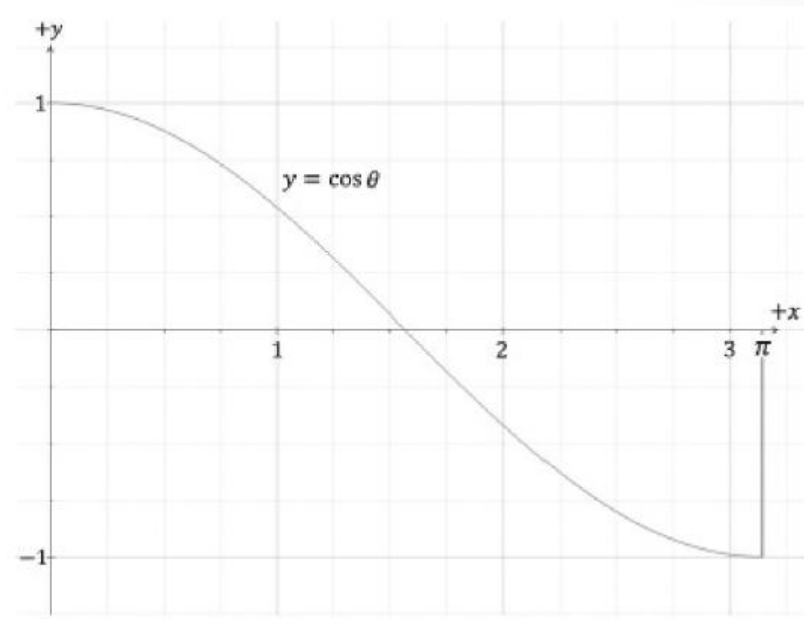
Now label the unit vector in the same direction as \mathbf{u} by \mathbf{n} : $\mathbf{n} = \mathbf{u} / \|\mathbf{u}\| = \mathbf{u} / \sin \theta$

Hence, $\mathbf{u} = \sin \theta \mathbf{n}$ and, the unit quaternion $\mathbf{q} = (\mathbf{u}, q_4)$ in the following *polar representation* where \mathbf{n} is a unit vector:

$$\mathbf{q} = (\sin \theta \mathbf{n}, \cos \theta) \text{ for } \theta \in [0, \pi]$$

Substituting $-\theta$ to θ is equivalent to negating the vector part of the quaternion:

$$(\mathbf{n} \sin(-\theta), \cos(-\theta)) = (-\mathbf{n} \sin \theta, \cos \theta) = \mathbf{p}^*$$



UNIT QUATERNIONS AND ROTATIONS

Let $\mathbf{q} = (\mathbf{u}, w)$ be a unit quaternion and let \mathbf{v} be a 3D point or vector. Then we can think

of \mathbf{v} as the pure quaternion $\mathbf{p} = (\mathbf{v}, 0)$. Also recall that since \mathbf{q} is a unit quaternion, we have that $\mathbf{q}^{-1} = \mathbf{q}^*$.

Recall the formula for quaternion multiplication:

$$(m, a)(n, b) = (an + bm + m \times n, ab - m \cdot n)$$

The quaternion rotation operator $R_{\mathbf{q}}(\mathbf{v}) = \mathbf{q}\mathbf{v}\mathbf{q}^{-1}$ rotates a vector (or point) \mathbf{v} about the axis \mathbf{n} by an angle 2θ .

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{v}) &= \mathbf{q}\mathbf{v}\mathbf{q}^{-1} \\ &= \mathbf{q}\mathbf{v}\mathbf{q}^* \\ &= \cos(2\theta)\mathbf{v} + (1 - \cos(2\theta))(\mathbf{n} \cdot \mathbf{v})\mathbf{n} + \sin(2\theta)(\mathbf{n} \times \mathbf{v}) \end{aligned}$$

So suppose you are given an axis \mathbf{n} and angle θ to rotate about the axis \mathbf{n} . You

construct the corresponding rotation quaternion by:

$$\mathbf{q} = \left(\sin\left(\frac{\theta}{2}\right)\mathbf{n}, \cos\left(\frac{\theta}{2}\right) \right)$$

Then apply the formula $R_{\mathbf{q}}(\mathbf{v})$. The division by 2 is to compensate for the 2θ because

we want to rotate by the angle θ , not 2θ .



Quaternion Rotation Operator to Matrix

Let $\mathbf{q} = (\mathbf{u}, w) = (q_1, q_2, q_3, q_4)$ be a unit quaternion.

$$R_{\mathbf{q}}(\mathbf{v}) = \mathbf{vQ} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} 1 - 2q_2^2 - 2q_3^2 & 2q_1q_2 + 2q_3q_4 & 2q_1q_3 - 2q_2q_4 \\ 2q_1q_2 - 2q_3q_4 & 1 - 2q_1^2 - 2q_3^2 & 2q_2q_3 + 2q_1q_4 \\ 2q_1q_3 + 2q_2q_4 & 2q_2q_3 - 2q_1q_4 & 1 - 2q_1^2 - 2q_2^2 \end{bmatrix}$$



Matrix to Quaternion Rotation Operator

Given the rotation Matrix, we want to find the

$$\mathbf{R} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix}$$

quaternion $\mathbf{q} = (q_1, q_2, q_3, q_4)$ such that if we build the matrix Q

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} = \begin{bmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4 \\ 2q_1q_2-2q_3q_4 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_1q_4 \\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 1-2q_1^2-2q_2^2 \end{bmatrix}$$

We start by summing the diagonal elements (which is called the *trace* of a matrix): from \mathbf{q} we get \mathbf{R} .

$$\begin{aligned} \text{trace}(\mathbf{R}) &= R_{11} + R_{22} + R_{33} \\ &= 1 - 2q_2^2 - 2q_3^2 + 1 - 2q_1^2 - 2q_3^2 + 1 - 2q_1^2 - 2q_2^2 \\ &= 3 - 4q_1^2 - 4q_2^2 - 4q_3^2 \\ &= 3 - 4(q_1^2 + q_2^2 + q_3^2) \\ &= 3 - 4(1 - q_4^2) \\ &= -1 + 4q_4^2 \\ \therefore q_4 &= \frac{\sqrt{\text{trace}(\mathbf{R}) + 1}}{2} \end{aligned}$$

Now we combine diagonally opposite elements to solve for q_1, q_2, q_3 (because we eliminate terms):

$$\begin{aligned} R_{23} - R_{32} &= 2q_2q_3 + 2q_1q_4 - 2q_2q_3 + 2q_1q_4 \\ &= 4q_1q_4 \\ \therefore &= \frac{R_{23} - R_{32}}{4q_4} \end{aligned}$$

$$\begin{aligned} R_{31} - R_{13} &= 2q_1q_3 + 2q_2q_4 - 2q_1q_3 + 2q_2q_4 \\ &= 4q_2q_4 \\ \therefore q_2 &= \frac{R_{31} - R_{13}}{4q_4} \end{aligned}$$

$$\begin{aligned} R_{12} - R_{21} &= 2q_1q_2 + 2q_3q_4 - 2q_1q_2 + 2q_3q_4 \\ &= 4q_3q_4 \\ \therefore q_3 &= \frac{R_{12} - R_{21}}{4q_4} \end{aligned}$$

Composition

Suppose \mathbf{p} and \mathbf{q} are unit quaternions
with corresponding rotational operators
given by $R_{\mathbf{p}}$ and $R_{\mathbf{q}}$, respectively.

Letting

$$\mathbf{v}' = R_{\mathbf{p}}(\mathbf{v})$$

$$R_{\mathbf{q}}(R_{\mathbf{p}}(\mathbf{v})) = R_{\mathbf{q}}(\mathbf{v}') = \mathbf{q}\mathbf{v}'\mathbf{q}^{-1} = \mathbf{q}(\mathbf{p}\mathbf{v}\mathbf{p}^{-1})\mathbf{q}^{-1} = (\mathbf{qp})\mathbf{v}(\mathbf{p}^{-1}\mathbf{q}^{-1}) = (\mathbf{qp})\mathbf{v}(\mathbf{qp})^{-1}$$

the composition is given by:

Because \mathbf{p} and \mathbf{q} are both unit
quaternions, the product \mathbf{pq} is also a
unit quaternion since $||\mathbf{pq}|| =$
 $||\mathbf{p}|| ||\mathbf{q}|| = 1$; thus, the quaternion
product \mathbf{pq} also represents a rotation



QUATERNION INTERPOLATION

Since quaternions are 4-tuples of real numbers, geometrically, we can visualize them as 4D vectors.

Unit quaternions are 4D unit vectors that lie on the 4D unit sphere.

With the exception of the cross product (which is only defined for 3D vectors), our vector math generalizes to 4-space—and even n -space.

Specifically, the dot product holds for quaternions. Let $\mathbf{p} = (u, s)$ and $\mathbf{q} = (v, t)$, then:

$$\mathbf{p} \cdot \mathbf{q} = u \cdot v + st = ||\mathbf{p}|| ||\mathbf{q}|| \cos \theta$$

Where θ is the angle between the quaternions. If the quaternions \mathbf{p} and \mathbf{q} are unit length, then $\mathbf{p} \cdot \mathbf{q} = \cos \theta$

The dot product allows us to talk about the angle between two quaternions, as a measure of how “close” they are to each other on the unit sphere.

For the purposes of animation, we want to interpolate from one orientation to another orientation. To interpolate quaternions, we want to interpolate on the arc of the unit sphere so that our interpolated quaternion is also a unit quaternion.

The following figure shows interpolating along the 4D unit sphere from \mathbf{a} to \mathbf{b} by an angle $t\theta$. The angle between \mathbf{a} and \mathbf{b} is θ , the angle between \mathbf{a} and \mathbf{p} is $t\theta$, and the angle between \mathbf{p} and \mathbf{b} is $(1 - t)\theta$.

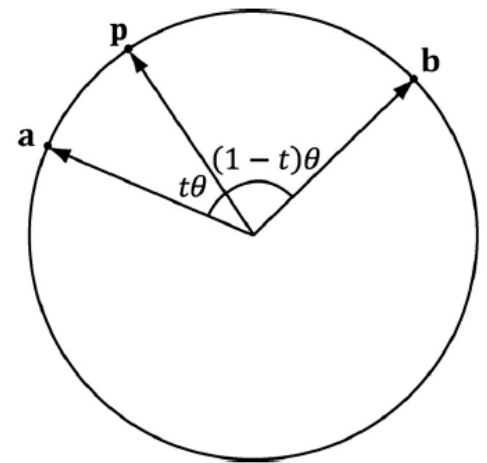
Thus we define the spherical interpolation formula:

$$\text{slerp}(\mathbf{a}, \mathbf{b}, t) = \frac{\sin((1-t)\theta)\mathbf{a} + \sin(t\theta)\mathbf{b}}{\sin \theta} \quad \text{for } t \in [0, 1]$$

Thinking of unit quaternions as 4D unit vectors allows us to solve for the angle between the quaternions:

$$\theta = \arccos(\mathbf{a} \cdot \mathbf{b}).$$

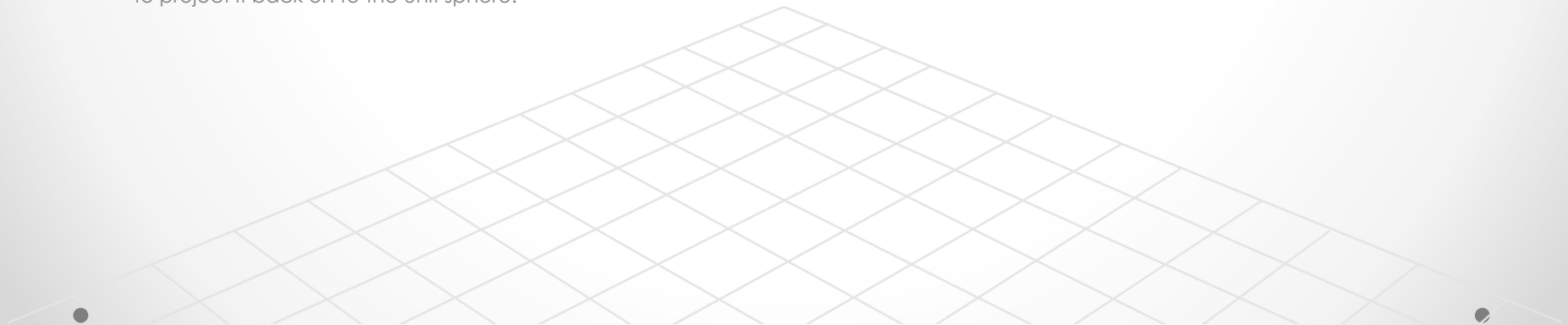
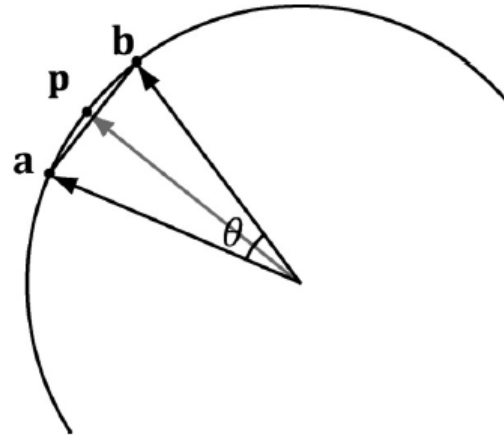
If θ , the angle between \mathbf{a} and \mathbf{b} is near zero, $\sin \theta$ is near zero, and the division can cause problems due to finite numerical precision. In this case, perform linear interpolation between the quaternions and normalize the result, which is actually a good approximation for small θ .



QUATERNION INTERPOLATION

For small angles θ between **a** and **b**, linear interpolation is a good approximation for spherical interpolation.

However, when using linear interpolation, the interpolated quaternion no longer lies on the unit sphere, so you must normalize the result to project it back on to the unit sphere.



QUATERNION INTERPOLATION

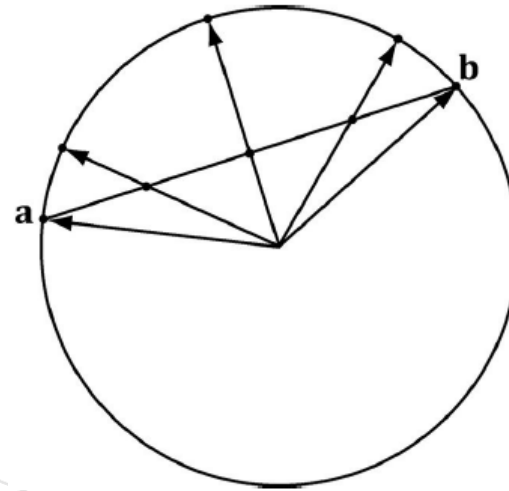
Notice that that linear interpolation followed by projecting the interpolated quaternion back on to the unit sphere results in a nonlinear rate of rotation.

we used linear interpolation for large angles, the speed of rotation will speed up and slow down.

This effect is often undesirable, and one reason why spherical interpolation is preferred (which rotates at a constant speed).

Linear interpolation results in nonlinear interpolation over the unit sphere after normalization.

This means the rotation speeds up and slows down as it interpolates, rather than moving at a constant speed.

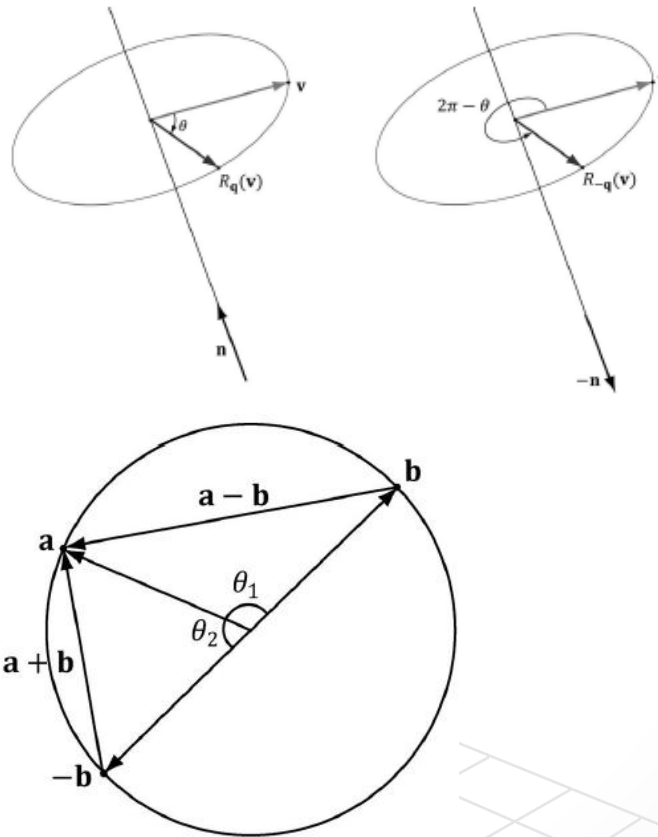


QUATERNION INTERPOLATION

R_q rotates θ about the axis \mathbf{n} , and R_{-q} rotates $2\pi - \theta$ about the axis $-\mathbf{n}$.

Interpolating from \mathbf{a} to \mathbf{b} results in interpolating over the larger arc θ_1 on the 4D unit sphere, whereas interpolating from \mathbf{a} to $-\mathbf{b}$ results in interpolating over the shorter arc θ_2 on the 4D unit sphere.

We want to choose the shortest arc on the 4D unit sphere.



LerpAndNormalize and Slerp functions

LerpAndNormalize and Slerp functions

// Linear interpolation (for small theta).

```
public static Quaternion
```

```
LerpAndNormalize(Quaternion p, Quaternion q,
```

```
float s)
```

```
{
```

// Normalize to make sure it is a unit

quaternion.

```
return Normalize((1.0f - s) * p + s * q);
```

```
}
```

```
public static Quaternion Slerp(Quaternion p,
Quaternion q, float s)
{
    // Recall that q and -q represent the same orientation, but
    // interpolating between the two is different: One will take the
    // shortest arc and one will take the long arc. To find
    // the shortest arc, compare the magnitude of p-q with the
    // magnitude p-(-q) = p+q.
    if (LengthSq(p - q) > LengthSq(p + q))
        q = -q;
    float cosPhi = DotP(p, q);
    // For very small angles, use linear interpolation.
    if (cosPhi > (1.0f - 0.001))
        return LerpAndNormalize(p, q, s);
    // Find the angle between the two quaternions.
    float phi = (float)Math.Acos(cosPhi);
    float sinPhi = (float)Math.Sin(phi);
    // Interpolate along the arc formed by the
    // intersection of the 4D
    // unit sphere and the plane passing through p, q,
    // and the origin of
    // the unit sphere.
    return ((float)Math.Sin(phi * (1.0 - s)) / sinPhi) * p +
        ((float)Math.Sin(phi * s) / sinPhi) * q;
}
```

DIRECTX MATH QUATERNION FUNCTIONS

The DirectX math library supports quaternions. Because the “data” of a quaternion is four real numbers, DirectX math uses the XMVECTOR type for storing quaternions. Then some of the common quaternion functions defined are:

```
// Returns the quaternion dot product Q1 · Q2.
XMVECTOR XMQuaternionDot(XMVECTOR Q1, XMVECTOR
Q2);
// Returns the identity quaternion (0, 0, 0, 1).
XMVECTOR XMQuaternionIdentity();
// Returns the conjugate of the quaternion Q.
XMVECTOR XMQuaternionConjugate(XMVECTOR Q);
// Returns the norm of the quaternion Q.
XMVECTOR XMQuaternionLength(XMVECTOR Q);
// Normalizes a quaternion by treating it as a 4D vector.
XMVECTOR XMQuaternionNormalize(XMVECTOR Q);
// Computes the quaternion product Q1Q2.
XMVECTOR XMQuaternionMultiply(XMVECTOR Q1, XMVECTOR
Q2);
```

```
// Returns a quaternions from axis-angle rotation representation.
XMVECTOR XMQuaternionRotationAxis(XMVECTOR Axis, FLOAT Angle);
// Returns a quaternions from axis-angle rotation representation, where the
axis
// vector is normalized—this is faster than XMQuaternionRotationAxis.
XMVECTOR XMQuaternionRotationNormal(XMVECTOR NormalAxis, FLOAT
Angle);
// Returns a quaternion from a rotation matrix.
XMVECTOR XMQuaternionRotationMatrix(XMMATRIX M);
// Returns a rotation matrix from a unit quaternion.
XMMATRIX XMMatrixRotationQuaternion(XMVECTOR Quaternion);
// Extracts the axis and angle rotation representation from the quaternion Q.
VOID XMQuaternionToAxisAngle(XMVECTOR *pAxis, FLOAT *pAngle,
XMVECTOR Q);
// Returns slerp(Q1, Q2, t)
XMVECTOR XMQuaternionSlerp(XMVECTOR Q0, XMVECTOR Q1, FLOAT t);
```

ROTATION DEMO

We animate a skull mesh around a simple scene.

The position, orientation, and scale of the mesh are animated.

We use quaternions to represent the orientation of the skull, and use slerp to interpolate between orientations.

We use linear interpolation to interpolate between position and scale.

A common form of animation is called key frame animation.

A *key frame* specifies the position, orientation, and scale of an object at an instance in time. In our demo (in *AnimationHelper.h/.cpp*), we define the following key frame structure:

```
Keyframe::Keyframe()
: TimePos(0.0f),
  Translation(0.0f, 0.0f, 0.0f),
  Scale(1.0f, 1.0f, 1.0f),
  RotationQuat(0.0f, 0.0f, 0.0f, 1.0f)
{
}

Keyframe::~~Keyframe()
{
}

float BoneAnimation::GetStartTime()const
{
  // Keyframes are sorted by time, so first keyframe gives start time.
  return Keyframes.front().TimePos;
}

float BoneAnimation::GetEndTime()const
{
  // Keyframes are sorted by time, so last keyframe gives end time.
  float f = Keyframes.back().TimePos;

  return f;
}

void BoneAnimation::Interpolate(float t, XMFLOAT4X4& M)const
{
}
```

BoneAnimation::Interpolate

We now have a list of key frames, which define the rough overall look of the animation.

How will the animation look at time between the key frames?

For times t between two key frames, say K_i and K_{i+1} , we interpolate between the two key frames K_i and K_{i+1} .

```
void BoneAnimation::Interpolate(float t,
XMLoadFloat4x4& M) const
{
    if( t <= Keyframes.front().TimePos )
    {
        XMVECTOR S =
XMLoadFloat3(&Keyframes.front().Scale);
        XMVECTOR P =
XMLoadFloat3(&Keyframes.front().Translation);
        XMVECTOR Q =
XMLoadFloat4(&Keyframes.front().RotationQuat);
```

```
        XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
        XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));
    }
    else if( t >= Keyframes.back().TimePos )
    {
        XMVECTOR S = XMLoadFloat3(&Keyframes.back().Scale);
        XMVECTOR P = XMLoadFloat3(&Keyframes.back().Translation);
        XMVECTOR Q = XMLoadFloat4(&Keyframes.back().RotationQuat);

        XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
        XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));
    }
    else
    {
        for(UINT i = 0; i < Keyframes.size()-1; ++i)
        {
            if( t >= Keyframes[i].TimePos && t <= Keyframes[i+1].TimePos )
            {
                float lerpPercent = (t - Keyframes[i].TimePos) / (Keyframes[i+1].TimePos -
                    Keyframes[i].TimePos);

                XMVECTOR s0 = XMLoadFloat3(&Keyframes[i].Scale);
                XMVECTOR s1 = XMLoadFloat3(&Keyframes[i+1].Scale);

                XMVECTOR p0 = XMLoadFloat3(&Keyframes[i].Translation);
                XMVECTOR p1 = XMLoadFloat3(&Keyframes[i+1].Translation);

                XMVECTOR q0 = XMLoadFloat4(&Keyframes[i].RotationQuat);
                XMVECTOR q1 = XMLoadFloat4(&Keyframes[i+1].RotationQuat);

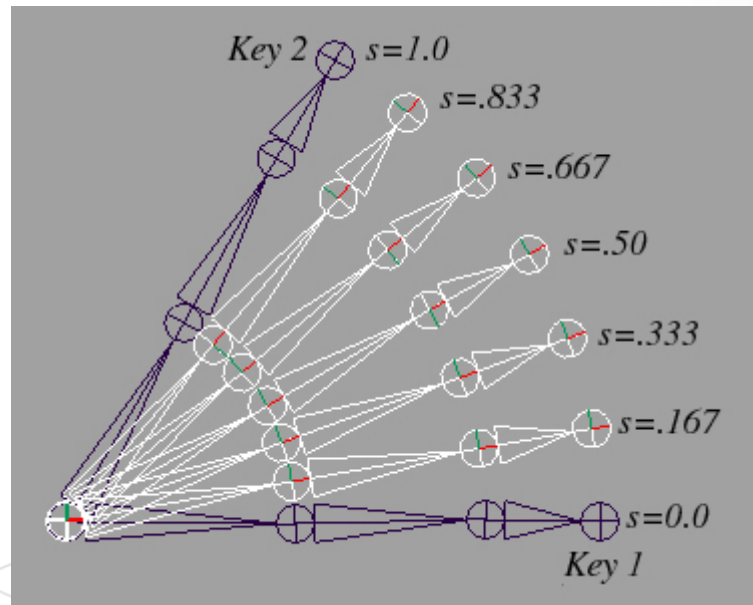
                XMVECTOR S = XMVectorLerp(s0, s1, lerpPercent);
                XMVECTOR P = XMVectorLerp(p0, p1, lerpPercent);
                XMVECTOR Q = XMQuaternionSlerp(q0, q1, lerpPercent);

                XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
                XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));
                break;}}}}
    }
```

Key frame interpolation

The key frames define the “key” poses of the animation. The interpolated values represent the values between the key frames.

Figure shows the in-between frames generated by interpolating from Key 1 to Key 2.



XMMatrixAffineTransformation

After interpolation, we construct a transformation matrix because ultimately we use matrices for transformations in our shader programs. The

XMMatrixAffineTransformation function is declared as follows:

XMMATRIX
XMMatrixAffineTransformation()
XMVECTOR Scaling,
XMVECTOR RotationOrigin,
XMVECTOR RotationQuaternion,
XMVECTOR Translation;

Now that our simple animation system is in place, the next part of our demo is to define some key frames:

```
// Member data
float mAnimTimePos = 0.0f;
BoneAnimation mSkullAnimation;
```

```
void QuatApp::DefineSkullAnimation()
{
    XMVECTOR q0 = XMQuaternionRotationAxis(XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(30.0f));
    XMVECTOR q1 = XMQuaternionRotationAxis(XMVectorSet(1.0f, 1.0f, 2.0f, 0.0f), XMConvertToRadians(45.0f));
    XMVECTOR q2 = XMQuaternionRotationAxis(XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(-30.0f));
    XMVECTOR q3 = XMQuaternionRotationAxis(XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f), XMConvertToRadians(70.0f));

    mSkullAnimation.Keyframes.resize(5);
    mSkullAnimation.Keyframes[0].TimePos = 0.0f;
    mSkullAnimation.Keyframes[0].Translation = XMFLOAT3(-7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[0].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[0].RotationQuat, q0);

    mSkullAnimation.Keyframes[1].TimePos = 2.0f;
    mSkullAnimation.Keyframes[1].Translation = XMFLOAT3(0.0f, 2.0f, 10.0f);
    mSkullAnimation.Keyframes[1].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[1].RotationQuat, q1);

    mSkullAnimation.Keyframes[2].TimePos = 4.0f;
    mSkullAnimation.Keyframes[2].Translation = XMFLOAT3(7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[2].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[2].RotationQuat, q2);

    mSkullAnimation.Keyframes[3].TimePos = 6.0f;
    mSkullAnimation.Keyframes[3].Translation = XMFLOAT3(0.0f, 1.0f, -10.0f);
    mSkullAnimation.Keyframes[3].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[3].RotationQuat, q3);

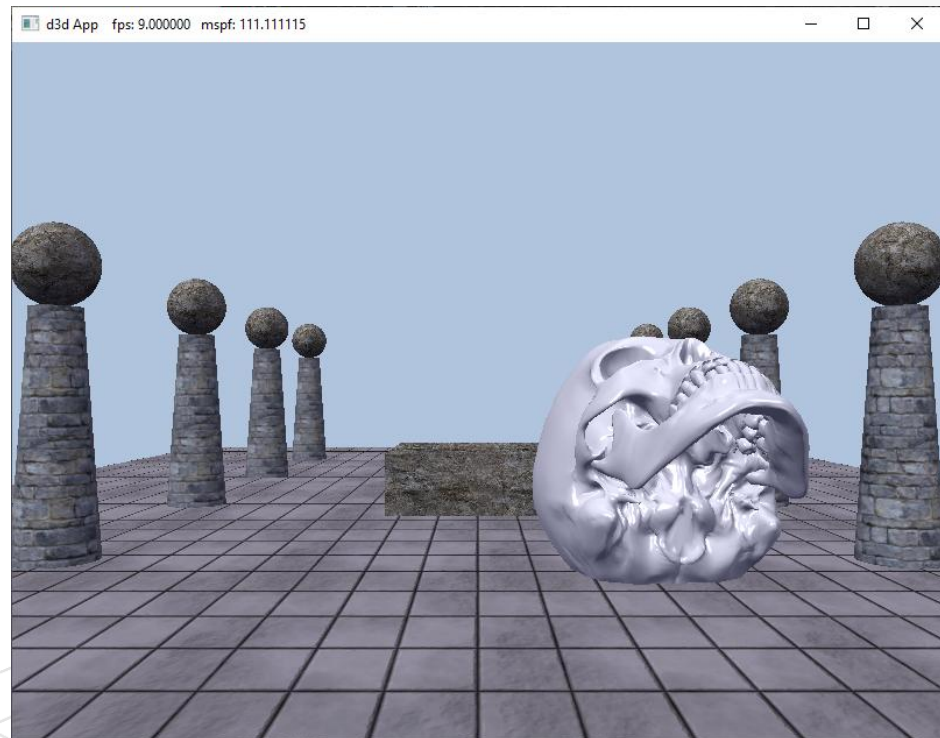
    mSkullAnimation.Keyframes[4].TimePos = 8.0f;
    mSkullAnimation.Keyframes[4].Translation = XMFLOAT3(-7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[4].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[4].RotationQuat, q0);
}
```


Key frames

Our key frames position the skull at different locations in the scene, at different orientations, and at different scales.

Have fun experimenting with this demo by adding your own key frames or changing the key frame values.

Set all the rotations and scaling to identity, to see what the animation looks like when only position is animated.



QuatApp::Update

The last step to get the animation working is to perform the interpolation to get the new skull world matrix, which changes over time:

```
void QuatApp::Update(const GameTimer& gt)
{
    OnKeyboardInput(gt);

    mAnimTimePos += gt.DeltaTime();
    if(mAnimTimePos >= mSkullAnimation.GetEndTime())
    {
        // Loop animation back to beginning.
        mAnimTimePos = 0.0f;
    }
```

```
mSkullAnimation.Interpolate(mAnimTimePos, mSkullWorld);
```

```
mSkullRitem->World = mSkullWorld;
```

```
mSkullRitem->NumFramesDirty = gNumFrameResources;
```

```
    // Cycle through the circular frame resource array.
    mCurrFrameResourceIndex = (mCurrFrameResourceIndex + 1) % gNumFrameResources;
    mCurrFrameResource = mFrameResources[mCurrFrameResourceIndex].get();

    // Has the GPU finished processing the commands of the current frame resource?
    // If not, wait until the GPU has completed commands up to this fence point.
    if(mCurrFrameResource->Fence != 0 && mFence->GetCompletedValue() < mCurrFrameResource-
    >Fence)
    {
        HANDLE eventHandle = CreateEventEx(nullptr, false, false, EVENT_ALL_ACCESS);
        ThrowIfFailed(mFence->SetEventOnCompletion(mCurrFrameResource->Fence, eventHandle));
        WaitForSingleObject(eventHandle, INFINITE);
        CloseHandle(eventHandle);
    }

    AnimateMaterials(gt);
    UpdateObjectCBs(gt);
    UpdateMaterialBuffer(gt);
    UpdateMainPassCB(gt);
}
```

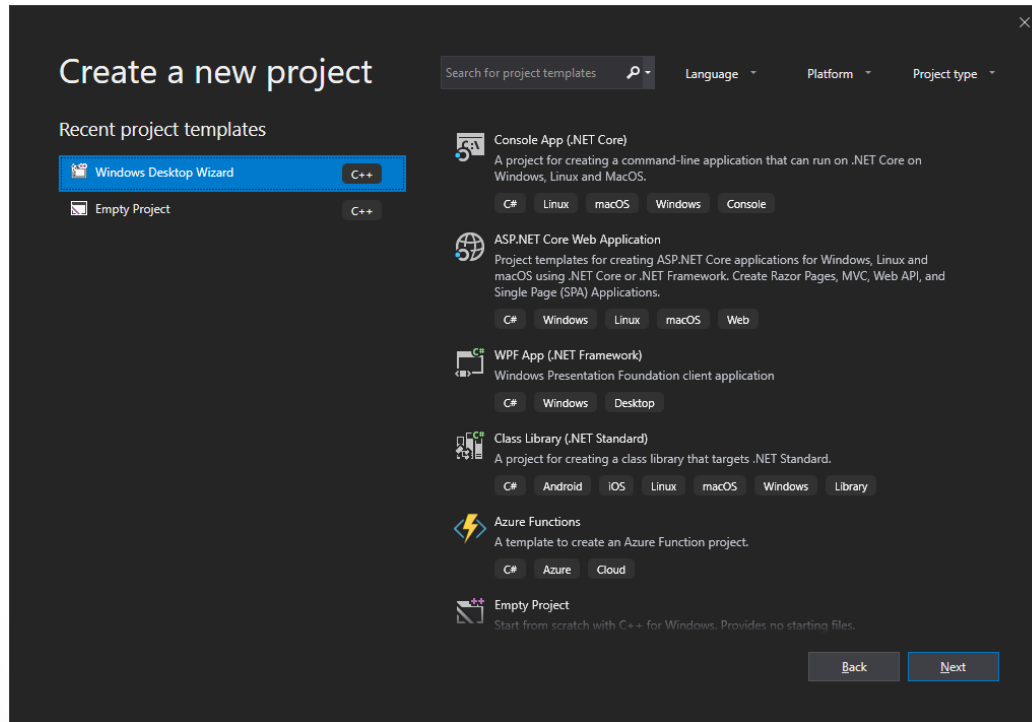
DirectX - Refresher

How to create a DirectX project

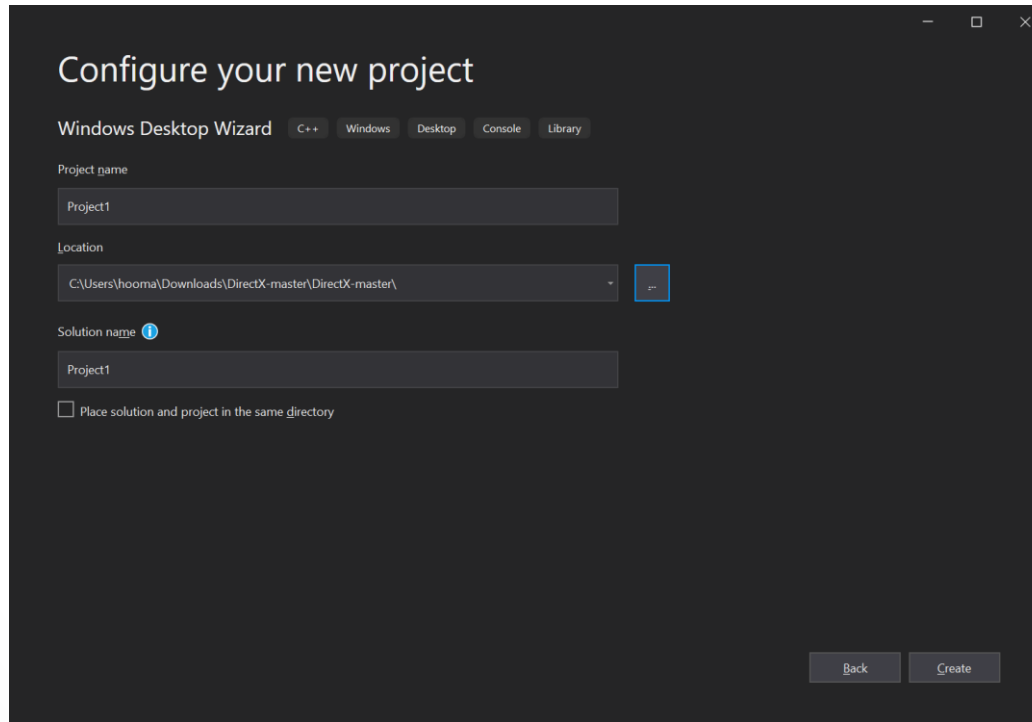
Demo

- Find “Widows Desktop Wizard” C++ project template and Create an empty Window Desktop project Make sure that you have “Solution1” and “Project1” folders are in the separate directory by not selecting the checkbox!
- Copy “Common” and “Texture” folders from GAME3015\GameEngineDevelopment2\Week1 next to your solution folder. Your “Solution” folder should be at the same level as Common and Textures

Create a new project in Visual Studio 2019/2022



Configure your new project



The screenshot shows a 'Configure your new project' dialog box with a dark theme. At the top, the title 'Configure your new project' is displayed. Below it, the 'Windows Desktop Wizard' is active, with tabs for 'C++', 'Windows', 'Desktop', 'Console', and 'Library'. The 'Project name' field contains 'Project1'. The 'Location' field shows the path 'C:\Users\hooma\Downloads\DirectX-master\DirectX-master\' with a browse button (three dots) to its right. The 'Solution name' field, which has an information icon, also contains 'Project1'. Below this, there is an unchecked checkbox labeled 'Place solution and project in the same directory'. At the bottom right, there are 'Back' and 'Create' buttons.

Configure your new project

Windows Desktop Wizard C++ Windows Desktop Console Library

Project name

Project1

Location

C:\Users\hooma\Downloads\DirectX-master\DirectX-master\

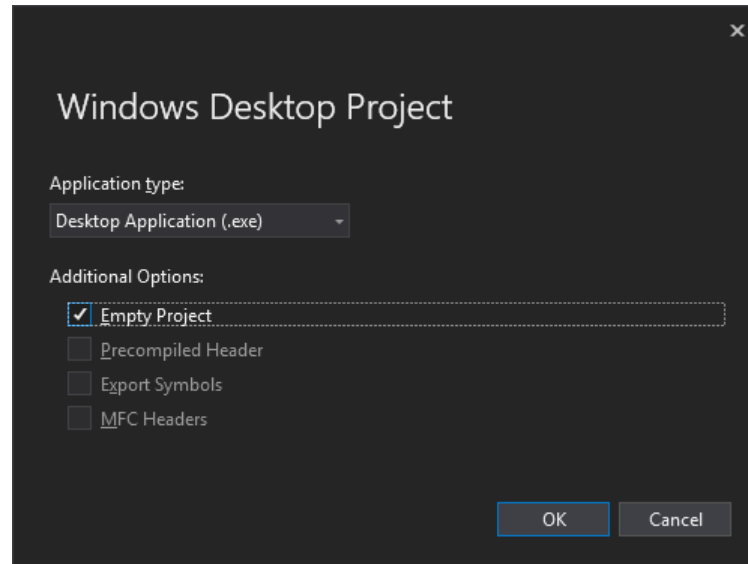
Solution name ⓘ

Project1

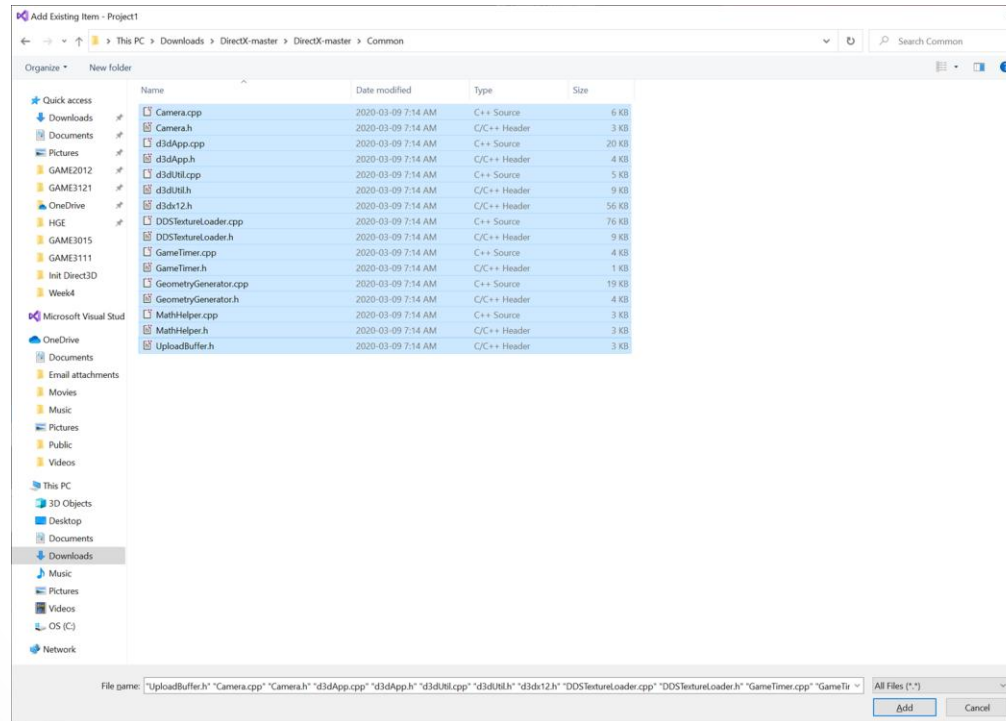
☐ Place solution and project in the same directory

Back Create

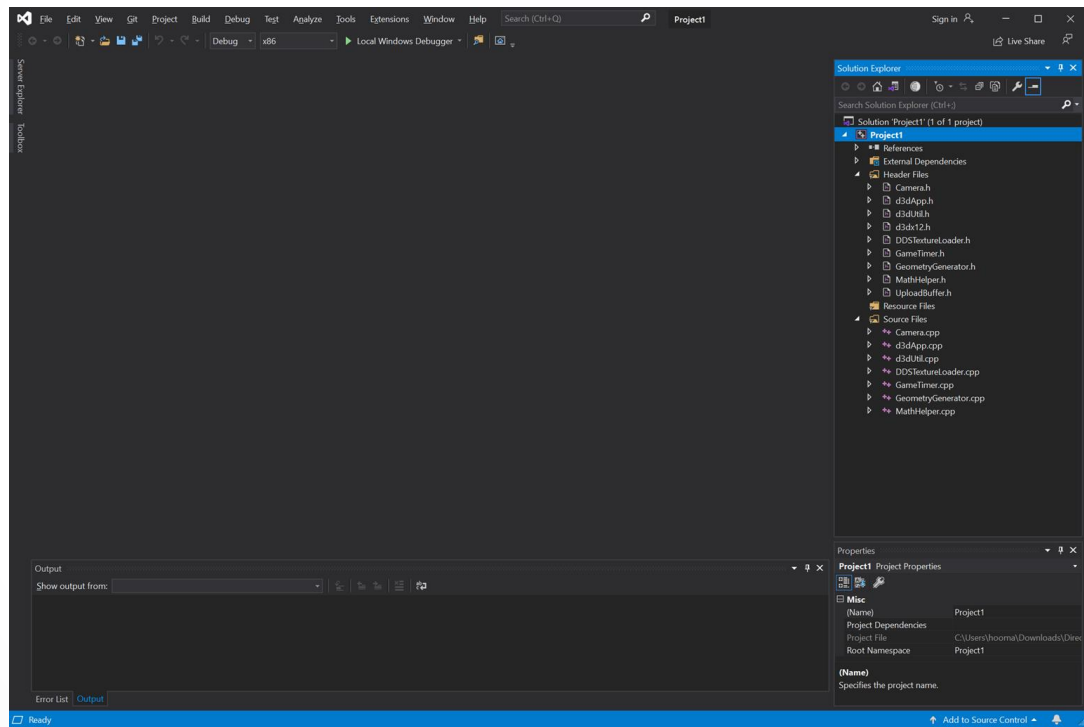
Create an Empty Windows Desktop Project



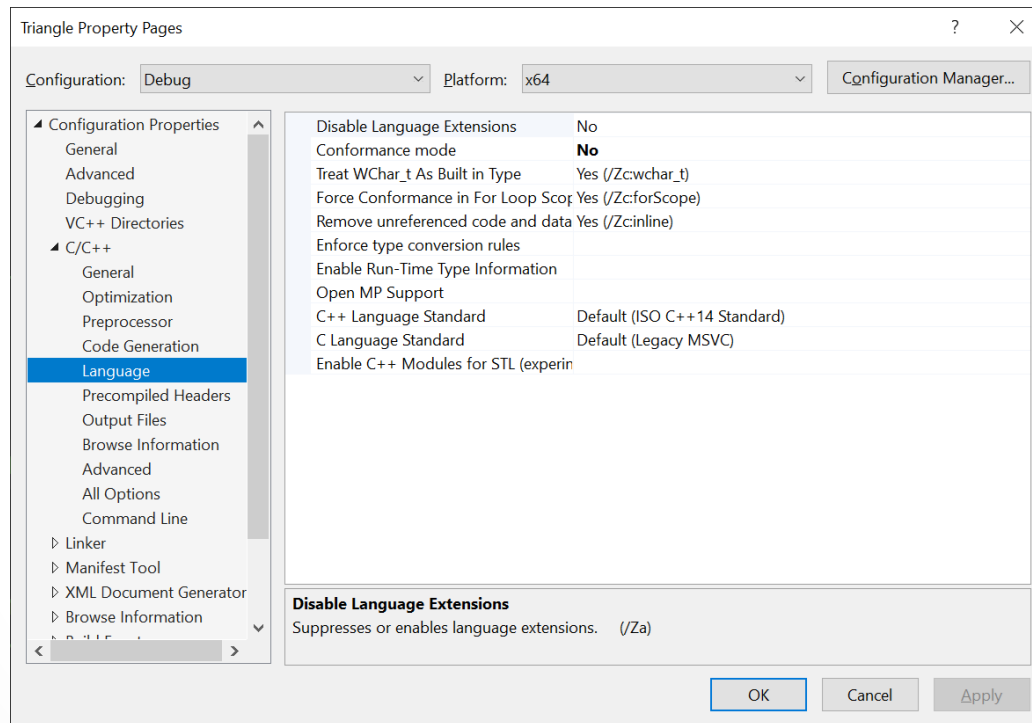
Add “Common” folder files Existing Item



Project1

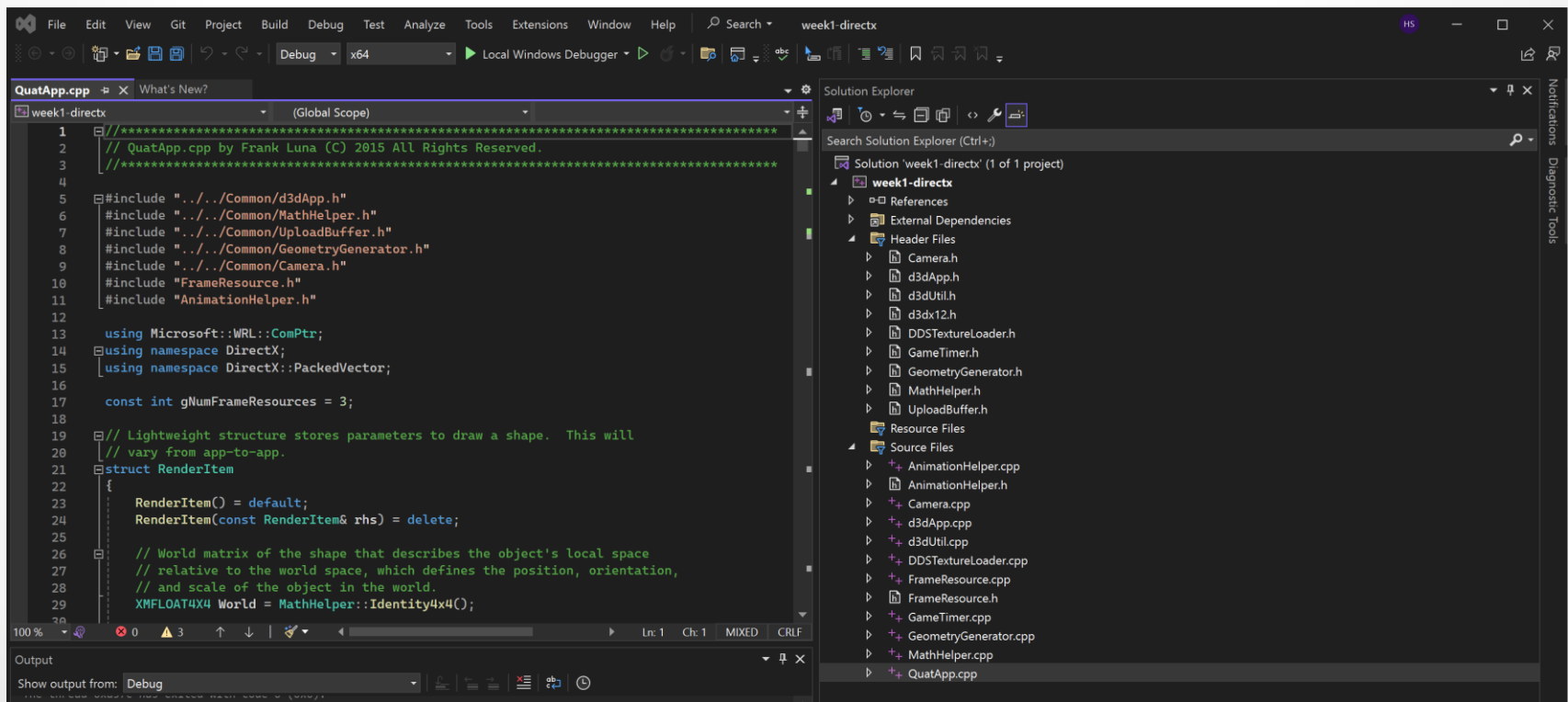


Change the conformance mode to “No”



Copy the QuantDemo files and place it under your project folder

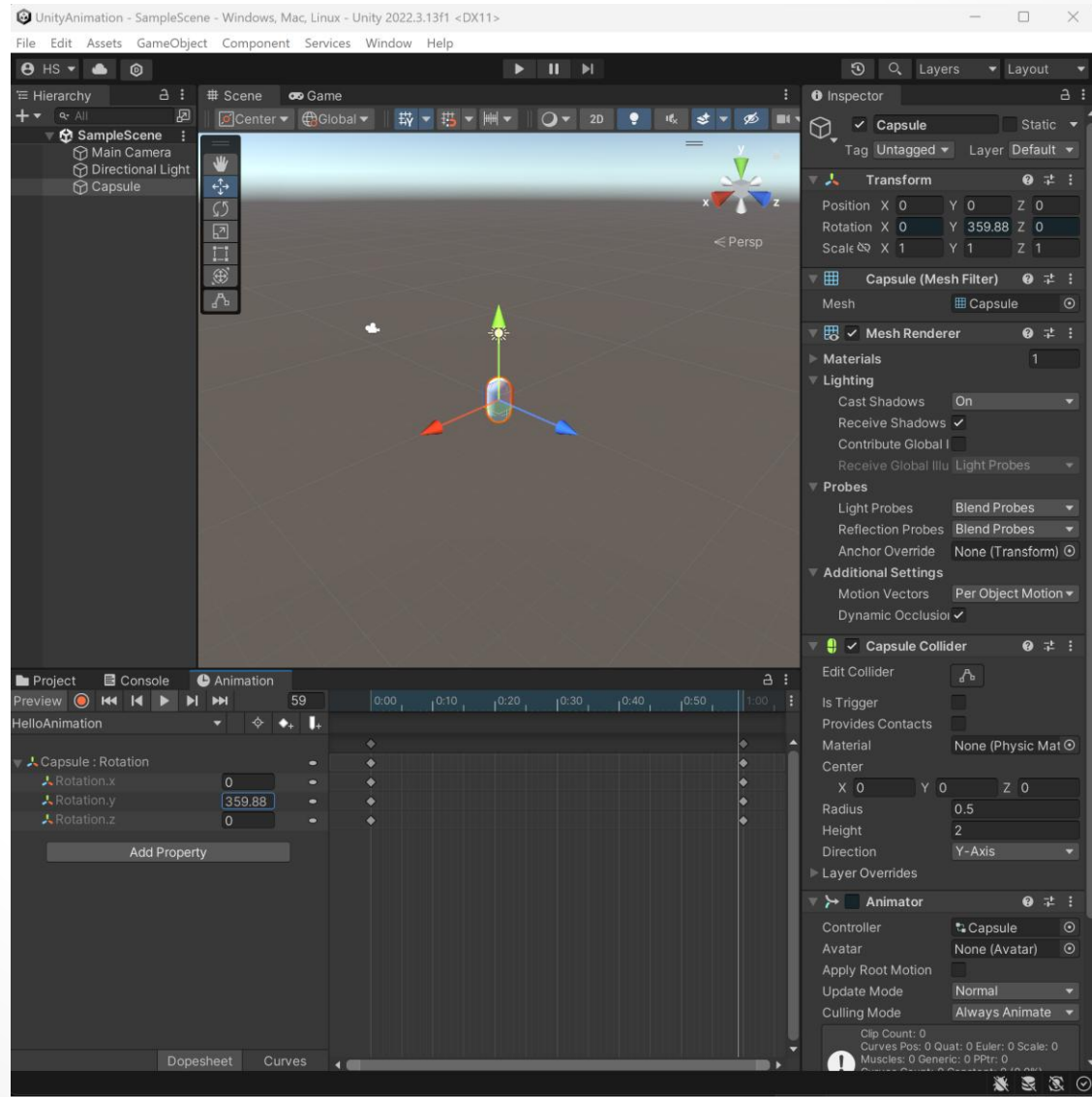
- GAME3015\GameEngineDevelopment2\Week1\QuatDemo



Animation In Unity

1. Add a capsule to the hierarchy
2. Select the "Capsule", go to Windows--> Animation --> Animation to open the animation window and then drag the animation window next to "Project" & "Console" at the bottom panel.
3. On animation window, click on "Create" button to create an Animator and Animation Clip
4. Name it "HelloAnimation.anim"
5. The first we need to do is to add the properties that we want to animate
6. Click "Add Property" in the Animation Window
7. Expand the "Transform" and add the "Rotation"
8. Now we have the key frames at the start and end of the animation
9. Key frames allows you to set the properties at the different times of the animation and Unity calculates all the values in between
10. Click on "Go to next key frame" icon (next to play button) to go to the last key frame
11. Open Capsule:Rotation and Change the Rotation.y to 360
12. Click on "Play" on Animation window to see the Capsule to spin

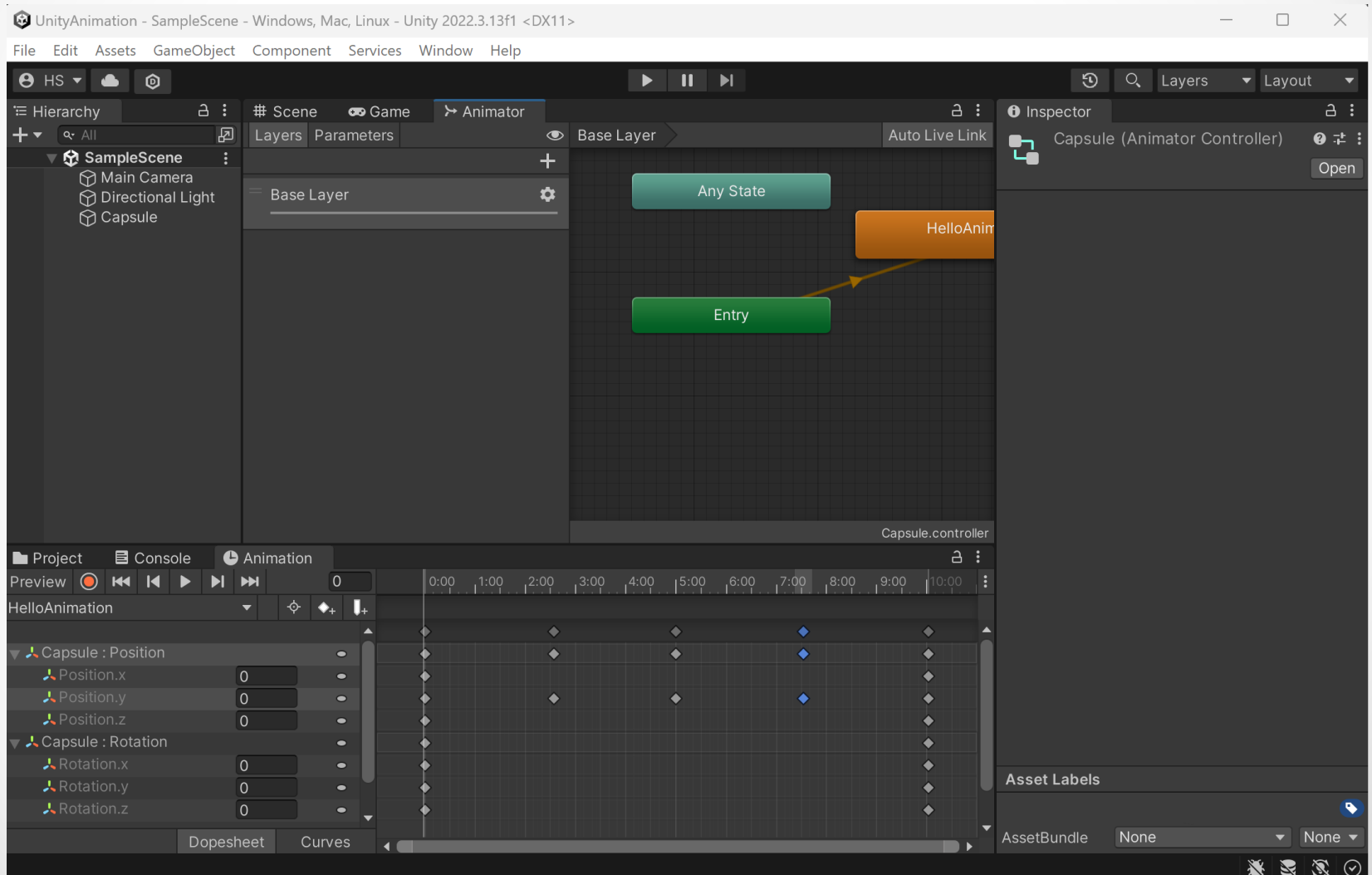
Key Frames



Duration

13. Right now the animation loops thru each second! May be too fast, let's change the duration
14. Use the middle mouse button to zoom out the time frame
15. Put your mouse on the last key frame, click and drag ot to 10 second
16. Now the animation loops every 10 seconds
17. Click on the "Curves" at the bottom of the "Animation" window
18. Notice that the curve slows down at the beginning or at the end of the curve!
19. On the curve, there are two diamonds that specifies the first and last key frame
20. Right click on each key frame, select "both tangent" --> Linear
21. Let's play the clip to check this
22. Now, we want our capsule to go up and down
23. Click on the Dopesheet, and click on "Add Property"
24. This time we select "Position" of the tranform component
25. Move the time to 2.5 second and change the Position.y to 0.5
26. Move the time to 5 second mark and set back the Position.y to 0.4
27. Move the time to 7.5 second mark and set the Position.y to 0.5
28. Play the clip again
29. Select the capsult in the hierarchy and "double" click on the "Capsule" Controller on "Animator" section in the inspector
30. Now you see the "Animator" window where "Entry" is connected to "HelloAnimation", which means the animation starts when the game starts.

Animator



Objectives

- Be introduced to SFML or Simple and Fast Multimedia Library, which is a C++ framework
- Learn how to download and install SFML
- Explore an example and see the format of an SFML program
- Examine the Game class of an SFML program

SFML Tutorials

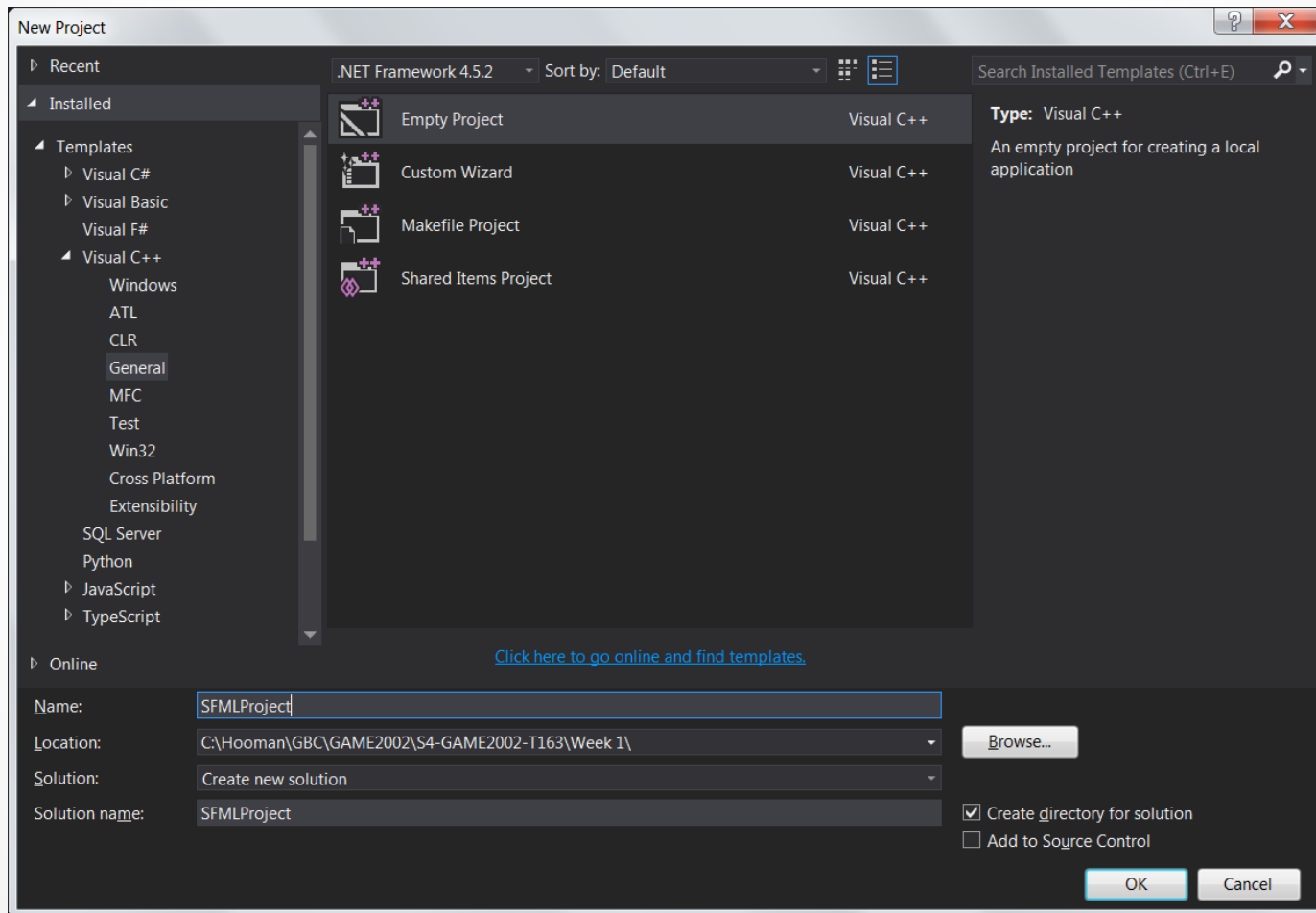
- Before we begin, here is a link to the main SFML tutorial site:
 - <https://www.sfml-dev.org/tutorials/2.5/>
 - <https://www.sfml-dev.org/tutorials/2.5/start-vc.php>
- Here you can also learn how to setup SFML for your version of Visual Studio – which we will go through in detail this week
- <http://sfml-hooman.blogspot.ca/2017/12/setting-up-sfml-242-in-visual-studio.html>

Setting up SFML with Visual Studio 2022

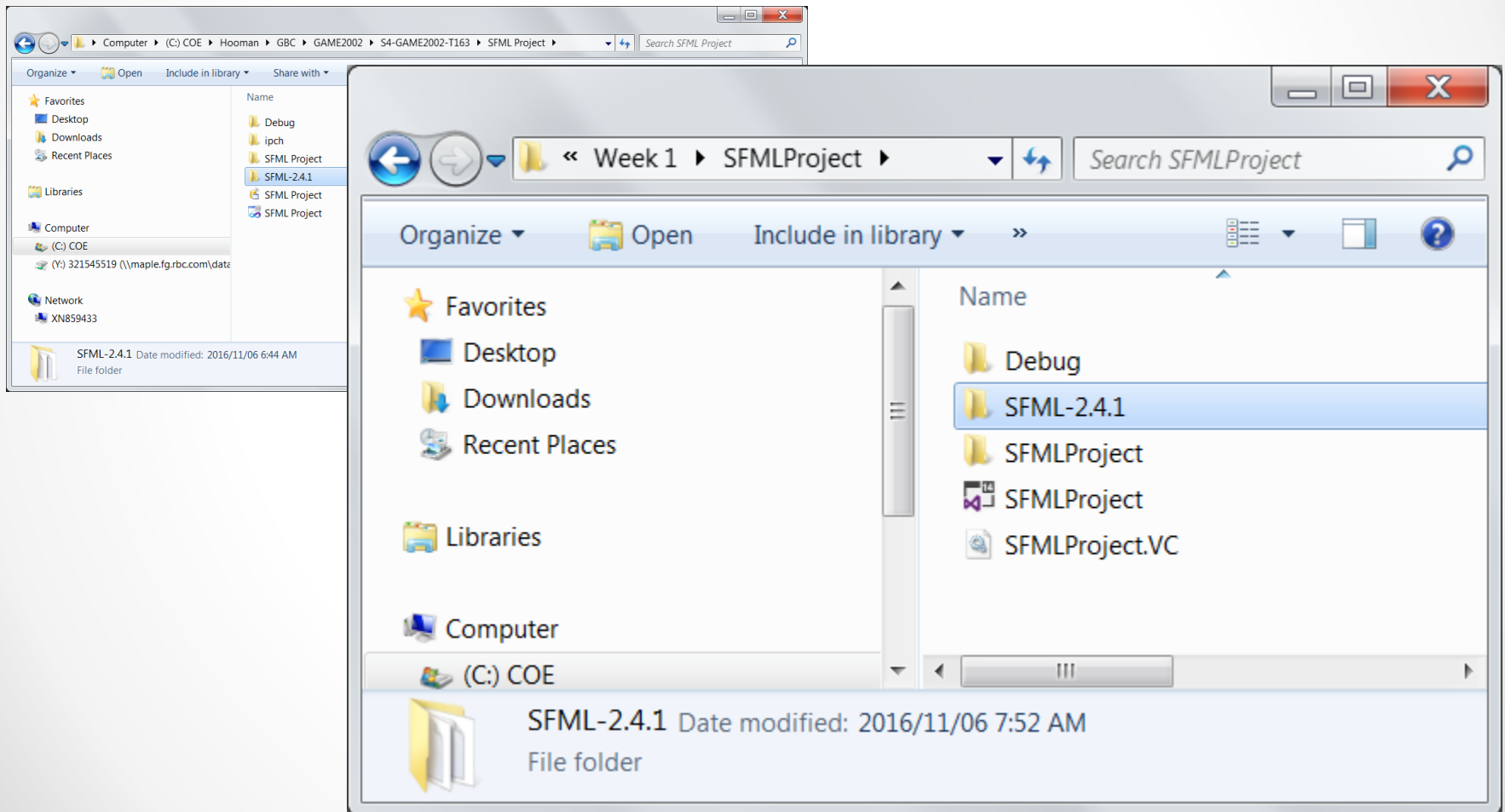
Installing SFML

- First, you must download the SFML SDK from the [download page](#).
- You must download the package that matches your version of Visual C++. Indeed, a library compiled with VC++ 10 (Visual Studio 2010) won't be compatible with VC++ 12 (Visual Studio 2013) for example. If there's no SFML package compiled for your version of Visual C++, you will have to [build SFML yourself](#).

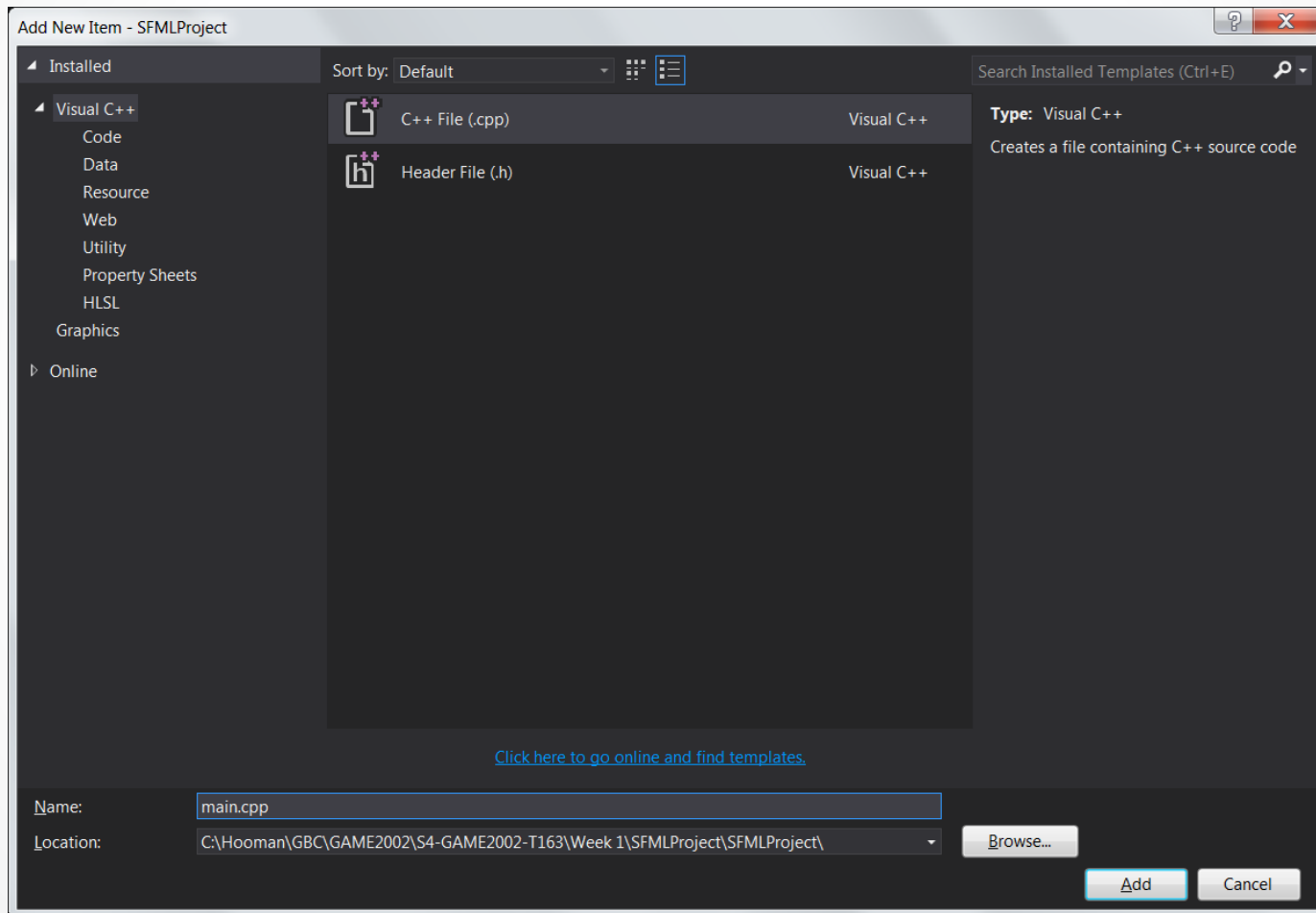
Create An Empty Project



Unzip and Copy SFML install folder under SFML Solution Directory

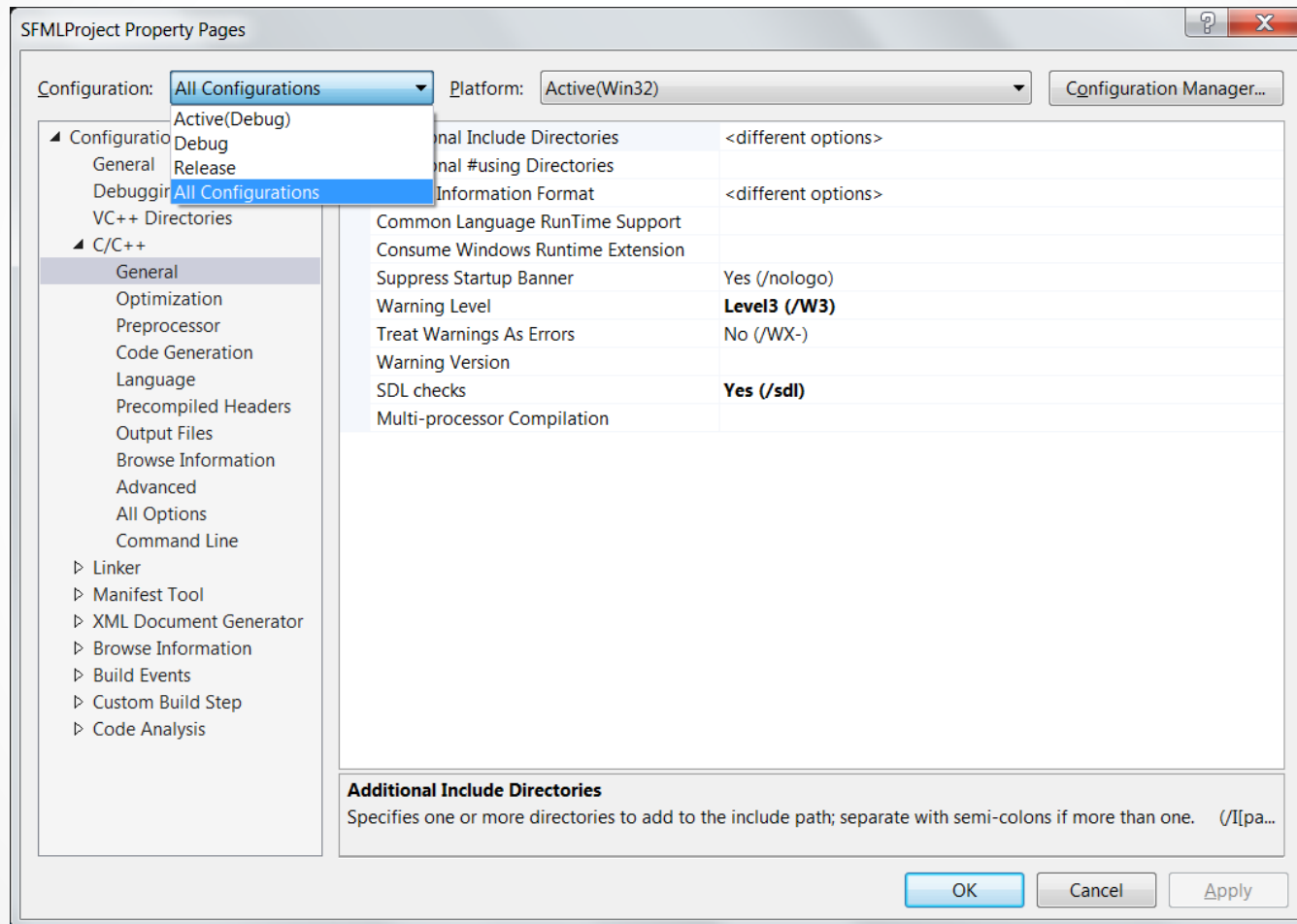


Add a C++ file to SFML project folder



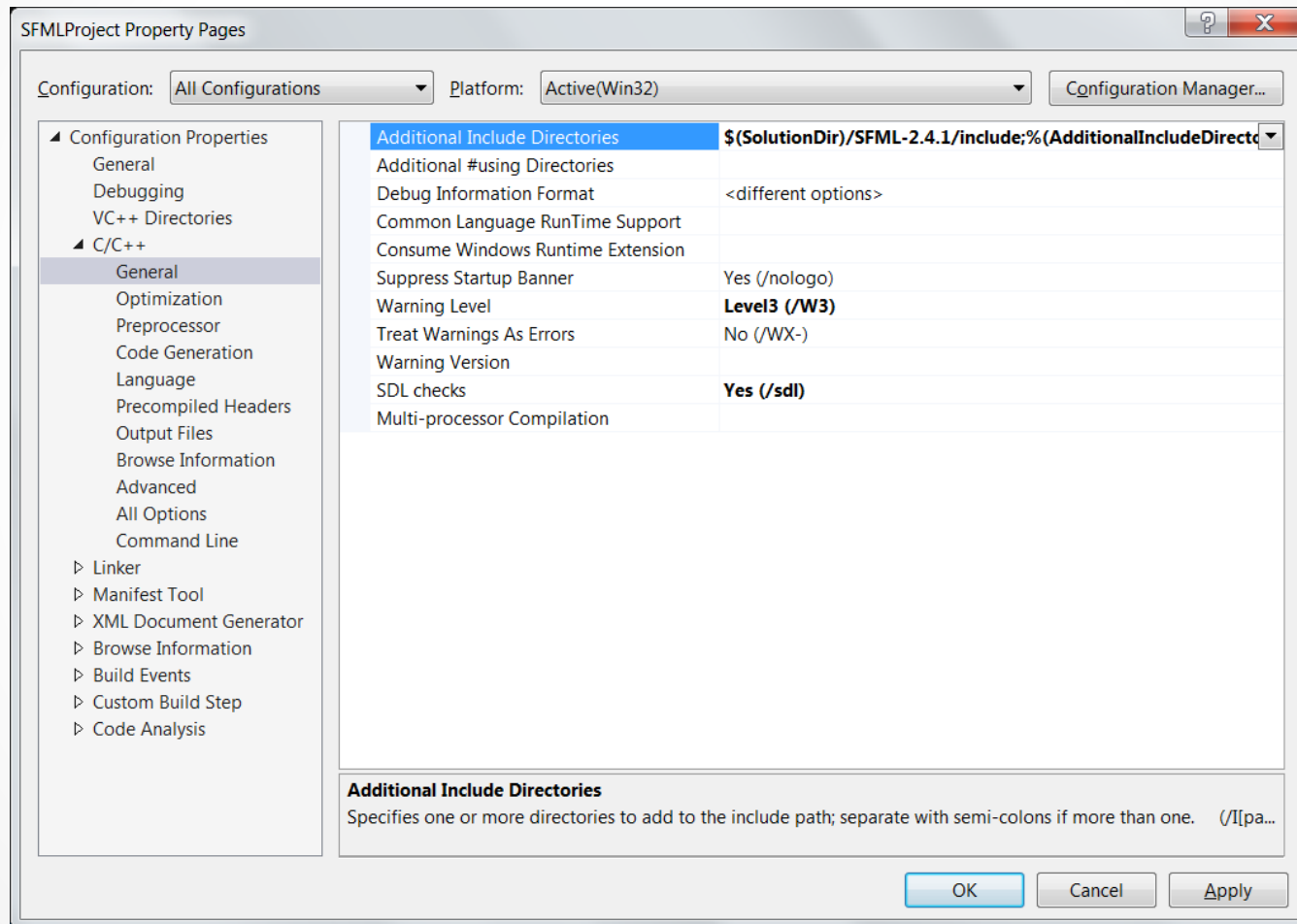
Go to Project Properties (Right Click on Project ☉ Select Project)

and Change the configuration to “All Configuration”

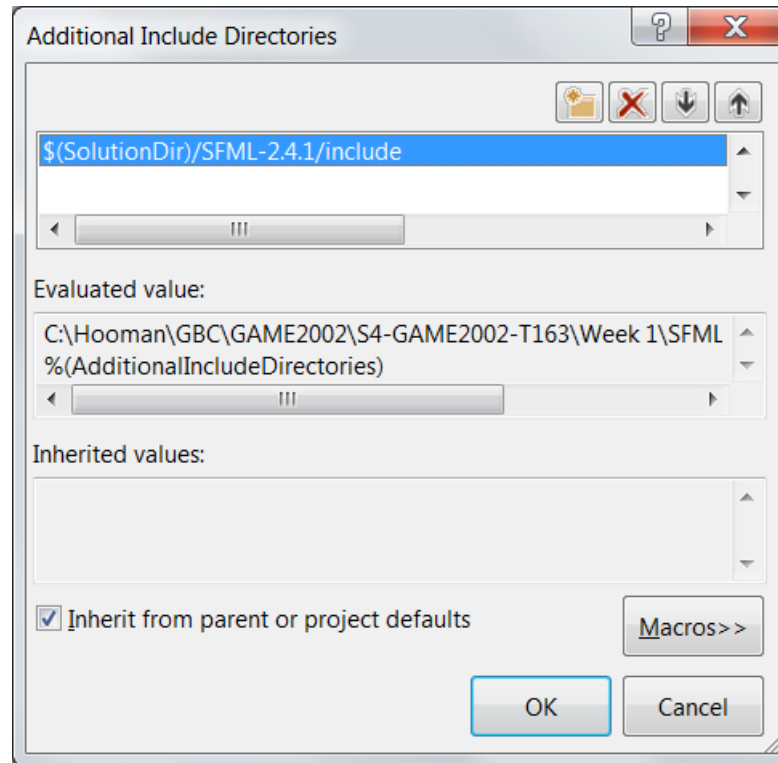


Add the SFML headers (*<sfml-install-path>/include*) to C/C++ »

General » Additional Include Directories

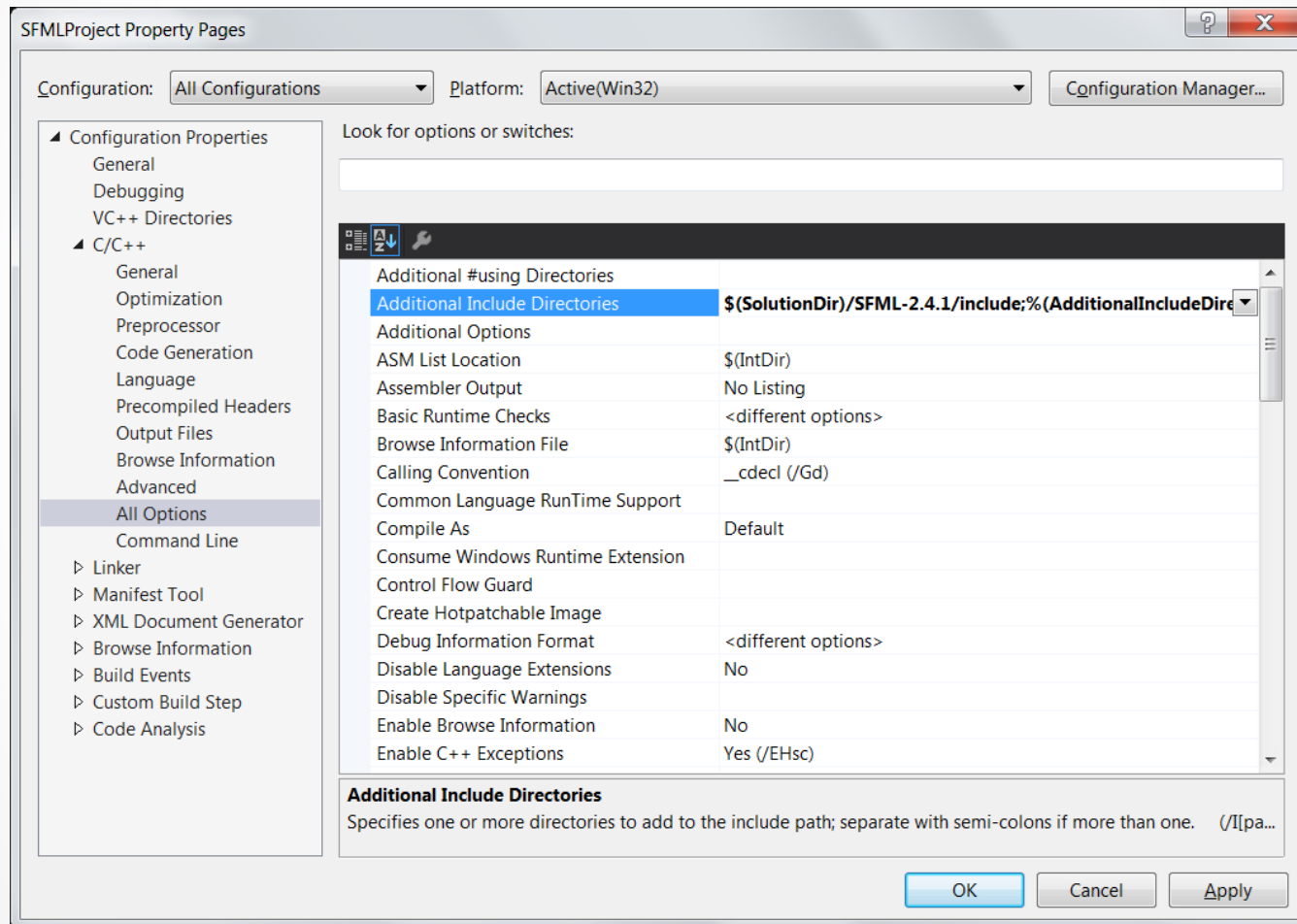


Additional Include Directories



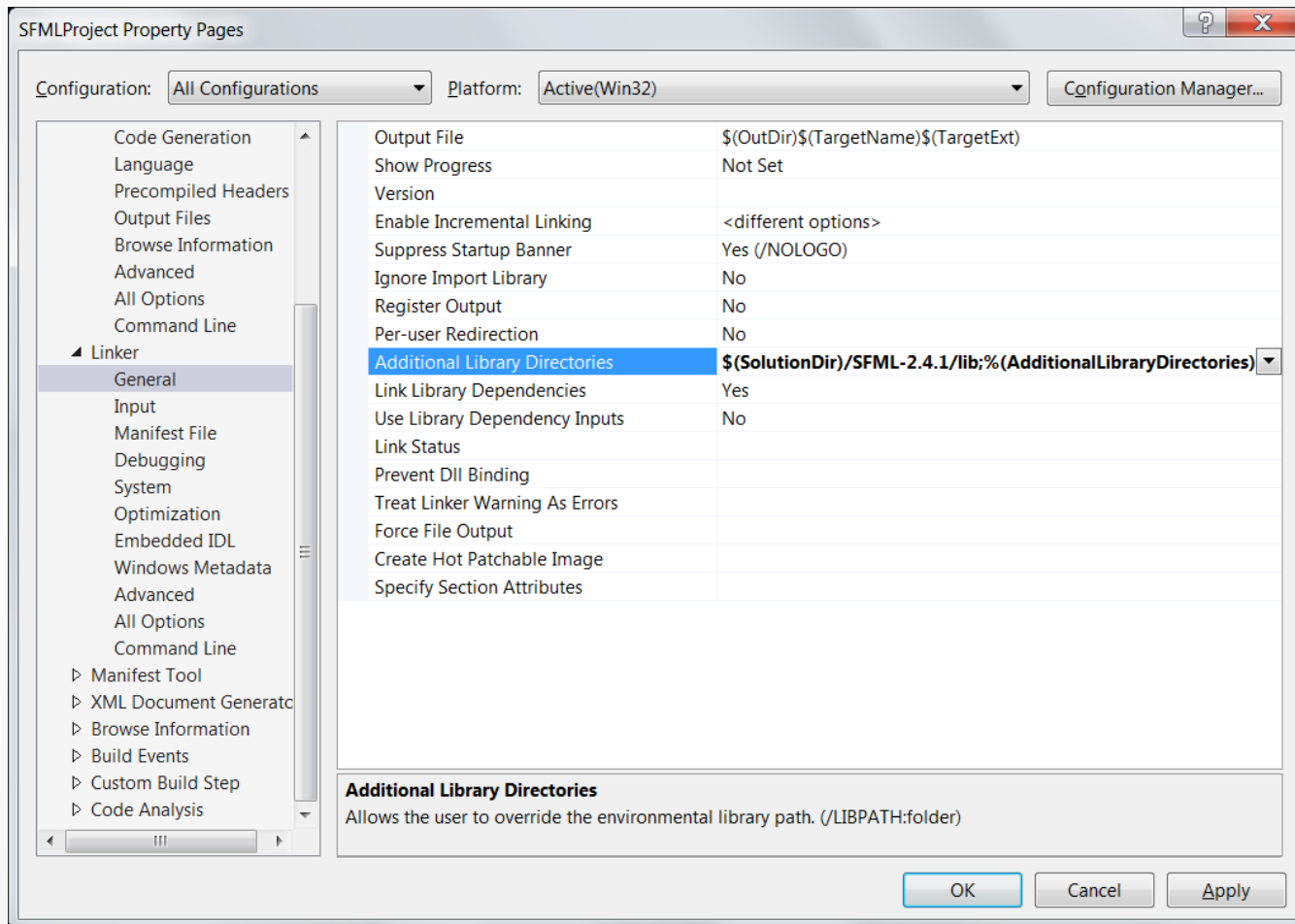
Add the SFML headers (*<sfml-install-path>/include*) to C/C++ » All Options »

Additional Include Directories

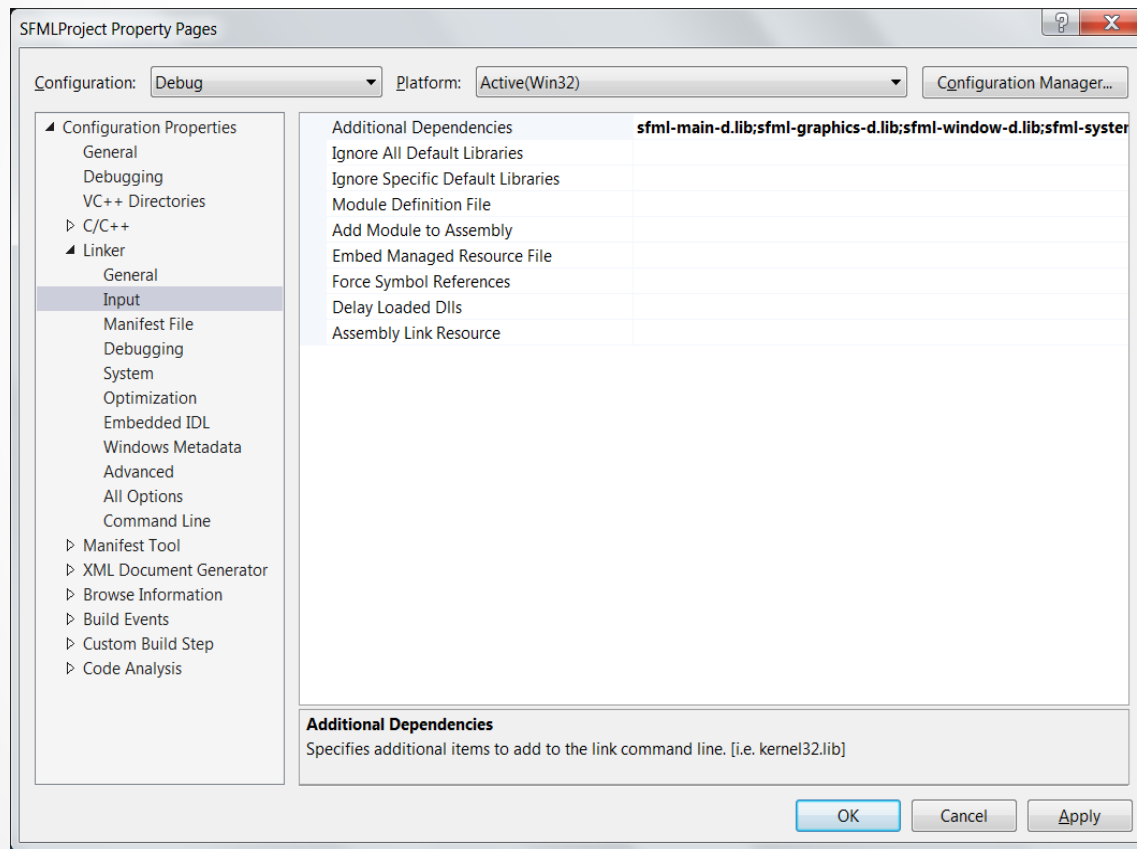


Add the SFML libraries (<sfml-install-path>/lib) to Linker » General » Additional

Library Directories

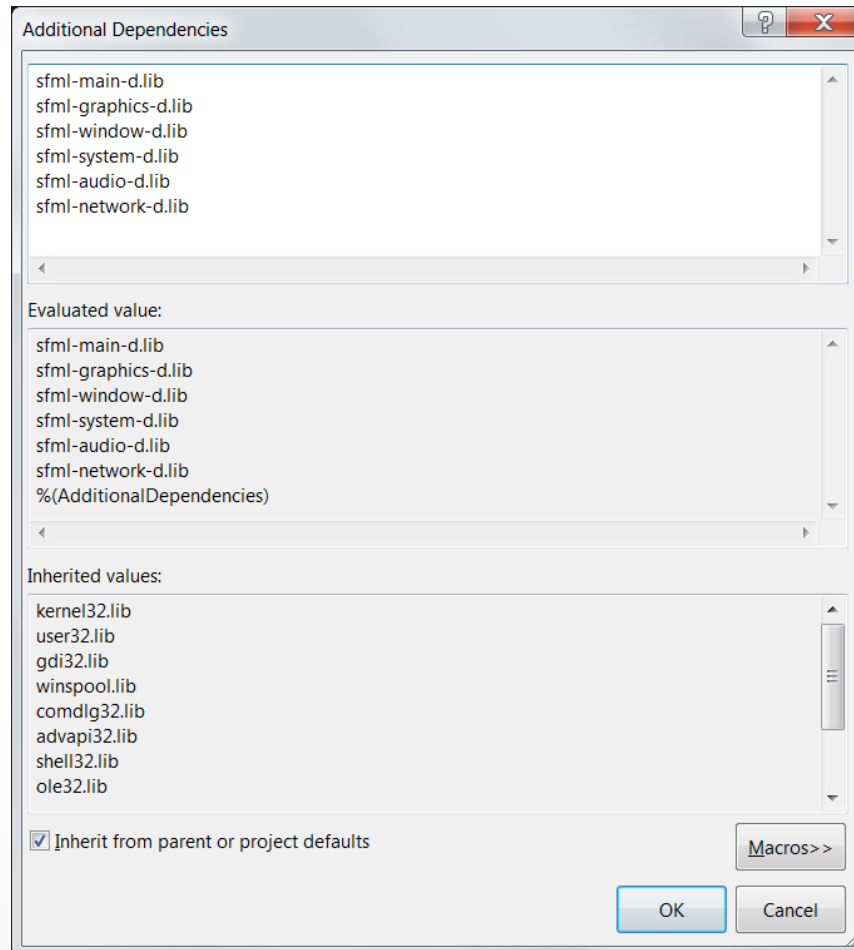


Add all the SFML libraries "sfml-graphics-d.lib", "sfml-window-d.lib" and "sfml-system-d.lib" in the project's properties, in Linker » Input » Additional Dependencies

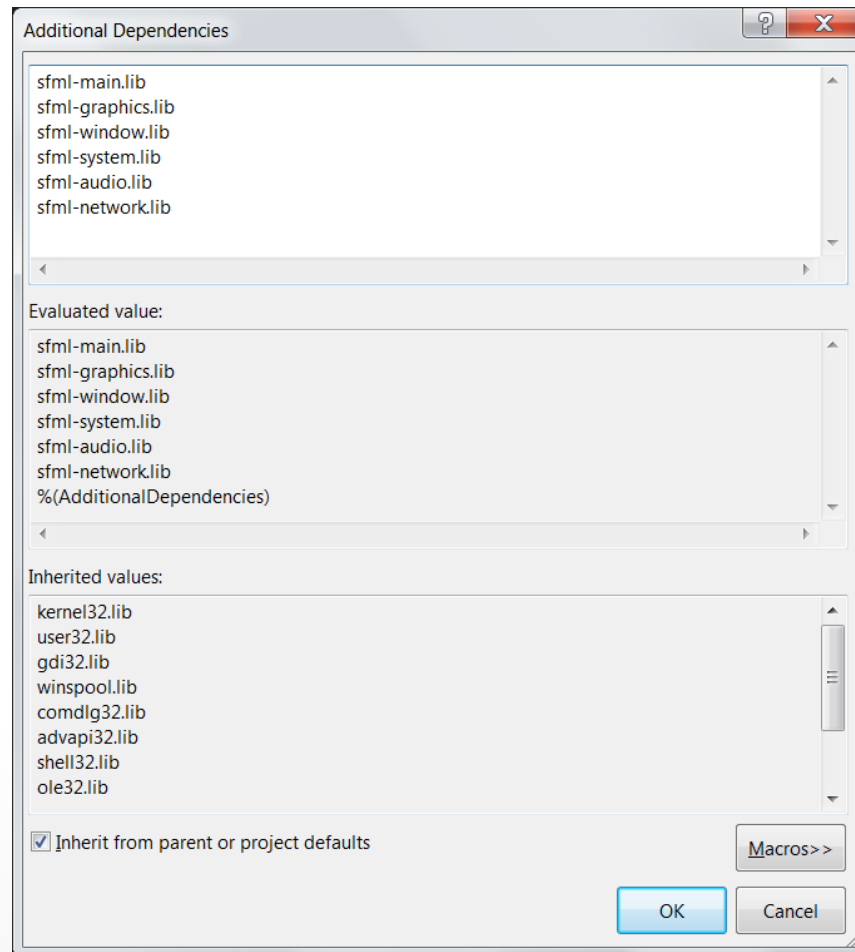


- It is important to link to the libraries that match the configuration: "sfml-xxx-d.lib" for Debug, and "sfml-xxx.lib" for Release.

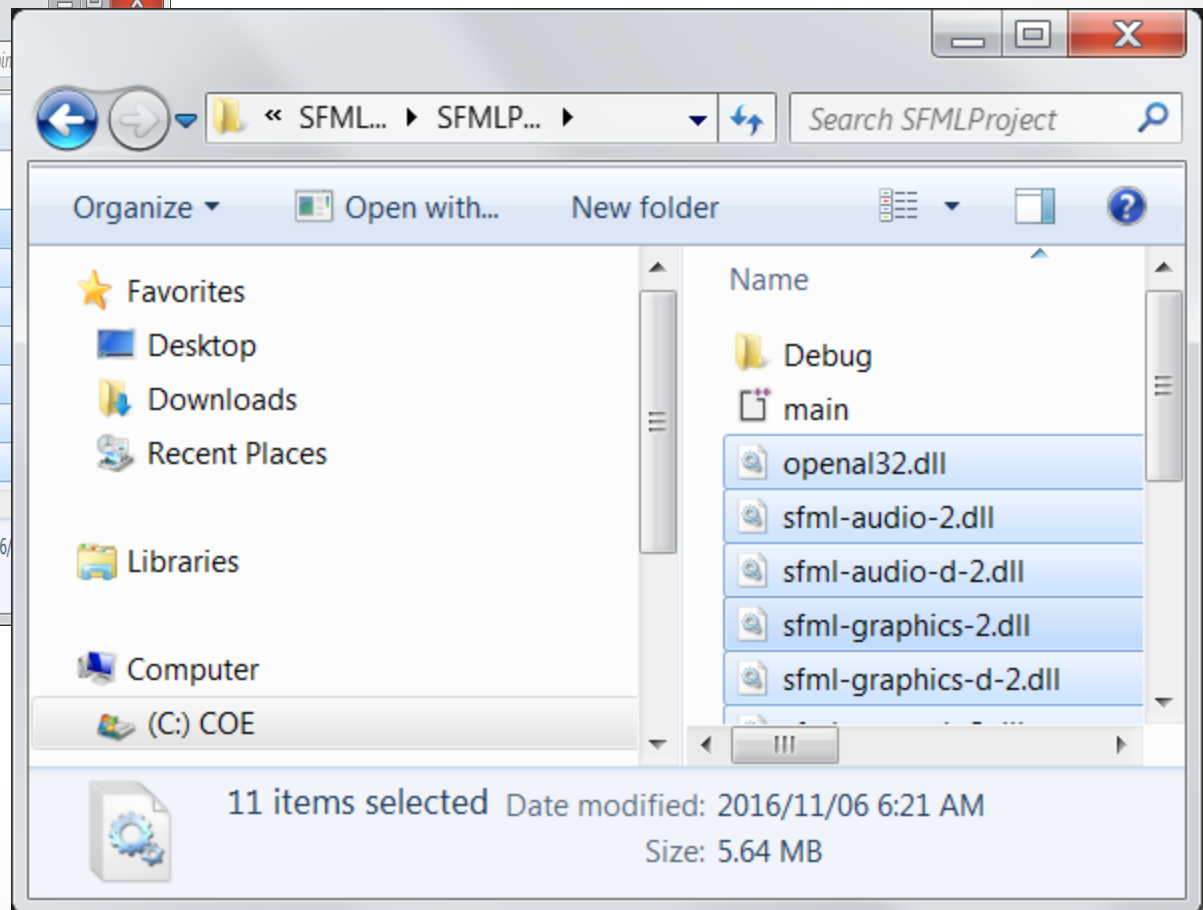
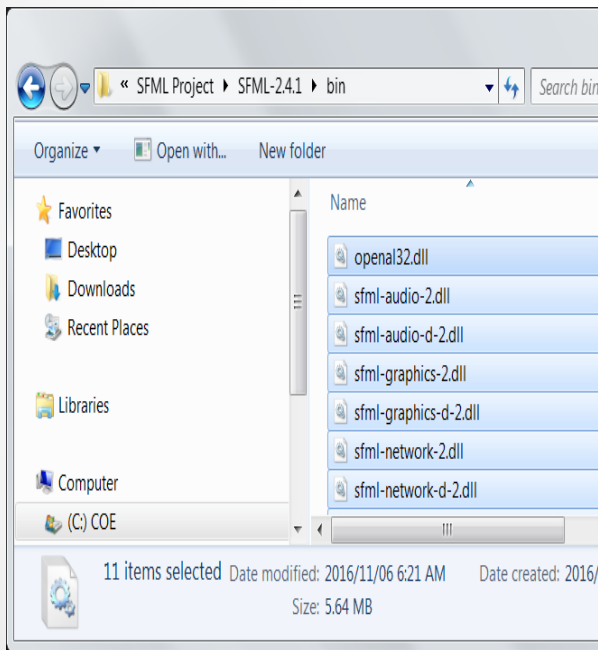
Additional Dependencies for Debug Configuration



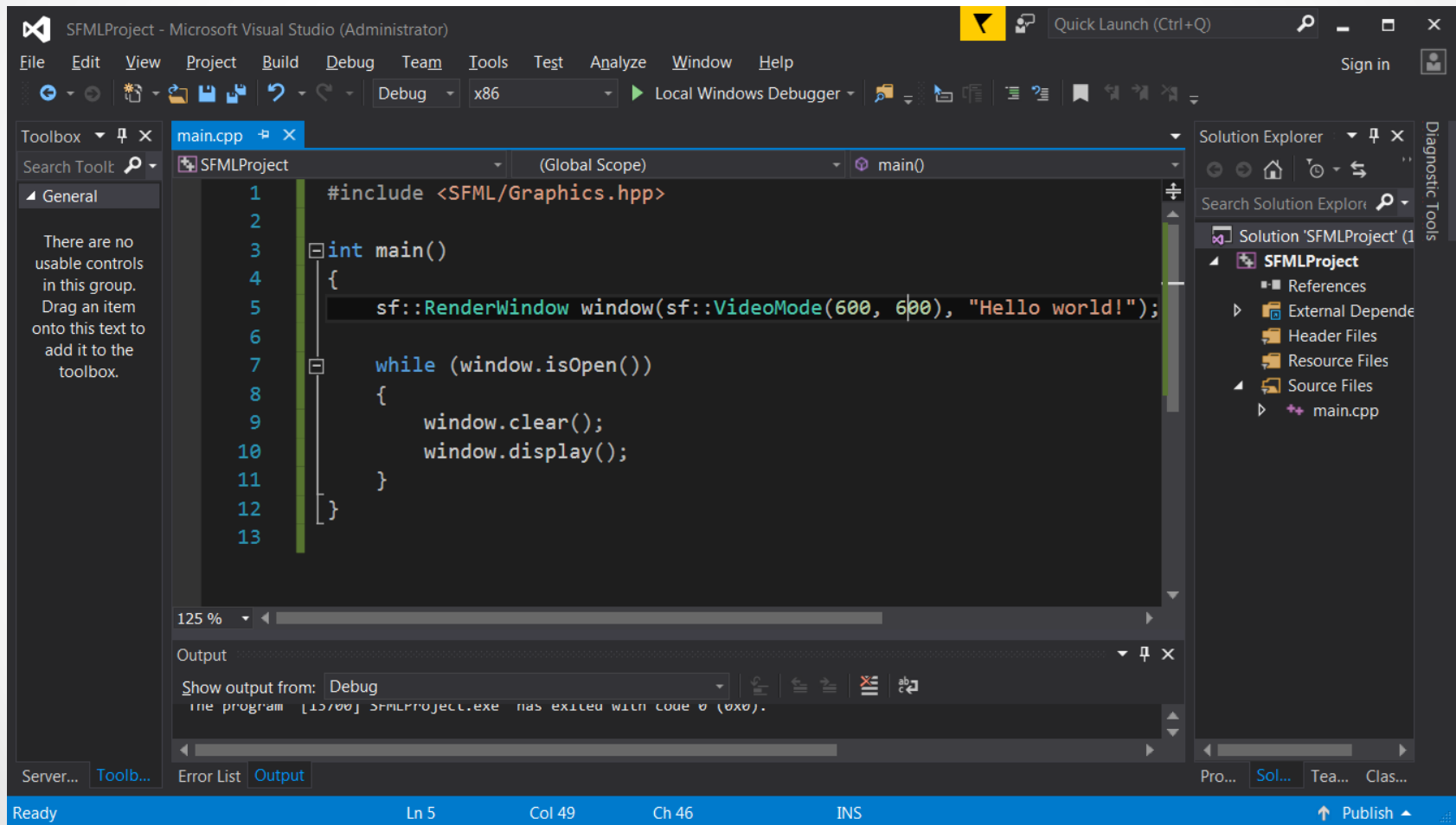
Additional Dependencies for Release Configuration



Copy all the dlls under (*<sfml-install-path>/bin*) to SFML project folder where
main.cpp is!



Put the following code inside the main.cpp file and Run



Replace the main.cpp with the following code

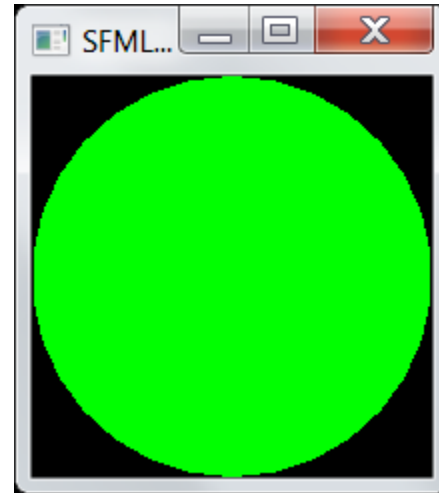
```
#include <SFML/Graphics.hpp>
```

```
int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }

    return 0;
}
```



Visual Studio Tips!

- If you choose to link the dynamic libraries, i.e.: *sfml-graphics.lib*, *sfml-window.lib* and *sfml-system.lib*, for **Release** or... *sfml-graphics-d.lib*, *sfml-window-d.lib* and *sfml-system-d.lib* for **Debug**...
 - **Do NOT add SFML_STATIC to the Preprocessor section**
 - **Remember to copy and paste the appropriate DLLs from bin to the same folder as your new .exe!**
- You can use `main()` instead of `WinMain()` even after choosing a Windows Application by including the appropriate *sfml-main.lib* or *sfml-main-d.lib* in the Linker->Input

Intro to SFML

- SFML is a library which adds multimedia content to your programs built in C++
- Five modules:
 - System
 - Window
 - Graphics
 - Audio
 - Network
- We'll start the course off by working with the first three for a few weeks

System Module

- The system is the core module
 - All other modules are built upon it
- It provides vector classes (2D and 3D), clocks, threads, Unicode strings and other things
- To use in your program:
 - Include sfml-system.lib in your Linker->Input
 - Or sfml-system-d.lib for Debug configuration

Window Module

- This module allows you to create application windows as well as collecting user input, such as mouse movement or key presses
 - You've seen Windows Application in Visual Studio before, but thus far your programs have been exclusively Console Applications
- To use in your program:
 - Include `sfml-window.lib` in your Linker->Input
 - Or `sfml-window-d.lib` for Debug configuration

Graphics Module

- The Graphics module allows you to include all functionality related to 2D rendering
 - Using images, texts, shapes and colors
- To use in your program:
 - Include `sfml-graphics.lib` in your Linker->Input
 - Or `sfml-graphics-d.lib` for Debug configuration

Audio Module

- The Audio module is, of course, provided so that you can add sounds to your game
 - Covers sound effects and music tracks
- To use in your program:
 - Include `sfml-audio.lib` in your Linker->Input
 - Or `sfml-audio-d.lib` for Debug configuration

Network Module

- Yes! SFML has a Network module that will allow you to setup multiplayer games
 - Includes everything you need to communicate over a LAN or the Internet using protocols such as HTTP and FTP
- And yes, we will be covering that in this course!
- To use in your program:
 - Include `sfml-network.lib` in your Linker->Input
 - Or `sfml-network-d.lib` for Debug configuration

SFML “Hello World”

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "Hello World!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }

    return 0;
}
```

Tips for Good Coding

- By this point, you should know how to code efficiently and use object-oriented features
- But let's reiterate some good concepts:
- Modularity
 - Keep your code separated into small pieces that perform a particular function
 - Separated into headers and implementation files
 - This will allow you to reuse that code easily, not only in the current program but other programs as well

Tips for Good Coding (cont'd.)

- Abstraction
 - Encapsulate functionality into classes and functions
 - This will prevent code duplications
 - Functions go way back to first term
- Consistency
 - Choose your coding style and stick to it so that it can be read easily and is more professional
 - Usually, this refers to how you use whitespace
 - Also how you use body braces, i.e.: { }

Abstraction into Practice

- To get you more familiar with SFML, we're going to take a minimal example on the next slide and convert the code into a class
- Through this, you should be able to see how we can break down the functionality into pieces and demonstrate how those pieces work together
- So let's get started!

Minimal Example

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(640,
    480), "SFML Application");
    sf::CircleShape shape;
    shape.setRadius(40.f);
    shape.setPosition(100.f, 100.f);
    shape.setFillColor(sf::Color::Cyan);
    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.clear();
        window.draw(shape);
        window.display();
    }
}
```

Game Class

```
class Game
{
    public:
        Game();
        void run();
    private:
        void processEvents();
        void update();
        void render();
    private:
        sf::RenderWindow mWindow;
        sf::CircleShape mPlayer;
};

int main()
{
    Game game;
    game.run();
}
```

Game Implementation

```
Game::Game()  
: mWindow(sf::VideoMode(640, 480), "SFML Application"), mPlayer()  
{  
    mPlayer.setRadius(40.f);  
    mPlayer.setPosition(100.f, 100.f);  
    mPlayer.setFillColor(sf::Color::Cyan);  
}  
  
void Game::run()  
{  
    while (mWindow.isOpen())  
    {  
        processEvents();  
        update();  
        render();  
    }  
}
```

Game Implementation (cont'd.)

```
void Game::processEvents()
{
    sf::Event event;
    while (mWindow.pollEvent(event))
    {
        if (event.type == sf::Event::Closed)
            mWindow.close();
    }
}

void Game::update()
{
}
```


DoxyGen

- DoxyWizard
- DoxyGen
- All assignments must be documented by doxygen!
- <https://www.doxygen.nl/index.html>
 - **Select JAVADOC_AUTOBRIEF** (in Project panel)
 - **Select EXTRACT_ALL** (in Build panel)
 - **DeSelect SHOW_USED_FILES**(in Build panel)
 - **Select Recursive**(in Input panel)
 - **Select sourceBrowser**(in SourceBrowser panel)
 - **Select DISABLE_INDEX** (in HTML panel)
 - **Select GENERATE_TREEVIEW** (in HTML panel)
 - **Create a README.dox**

Tags

```
/** @file passbyDemo.cpp
 *  @brief difference in passing in a variable
 *  by ref/value/pointer to a function
 *  @author Hooman Salamat
 *  @bug No known bugs.
 */
```

Important: file name must match the actual file name!

Example

```
/** @file passbyDemo.cpp
 *  @brief difference in passing in a variable by
 *  ref/value/pointer to a function
 *  @author Hooman Salamat
 *  @bug No known bugs.
 */

#include <cstdio>
#include <string>
#include <stdio.h>
#include <iostream>
using namespace std;
void passByVal(int val); //pass in a copy of the
variable
void passByRef(int& ref); //pass in the actual variable
void passByPtr(int* ptr); //pass in the address of the
variable

int main()
{
    string bye;
    int val = 5;
    passByVal(val);
    passByRef(val);
    passByPtr(&val);

    getline(cin, bye);
    return 0;
}
```

```
void passByVal(int val)
{
    val = 10;
    printf("val = %i \n", val);
}
```

```
void passByRef(int& ref)
{
    ref = 20;
    printf("ref = %i \n", ref);
}
```

```
void passByPtr(int* ptr)
{
    printf("*ptr = %i \n", *ptr);
    *ptr = 30;
    printf("*ptr = %i \n", *ptr);
}
```

README.dox example

/**

@mainpage assignment1

@author Hooman Salamat

@attention use WASD to move the paddle

@note this app doesn't work with mouse or arrows

this is my assignment 1. In this assignment, I am implementing

*/

Create your own repository for assignments and invite me and your partners as collaborators

1. Ask for the username of the person you're inviting as a collaborator. ...
2. On GitHub, navigate to the main page of the repository.
3. Under your repository name, click Settings.
4. In the left sidebar, click Manage access.
5. Click Invite a collaborator.