Institut Sabadell

Generalitat de Catalunya

Departament d'Ensenyament

Unió Europea

Fons Social Europeu

CICLE FORMATIU GRAU SUPERIOR

DESENVOLUPAMENT D'APLICACIONS MULTIPLATAFORMA

Manual tècnic

Curs: 2012-2013

Memòria Presentada per:

Alberto Lopez Sanchez

Ruben Bagan Benavides

Sabadell, 31 de Maig de 2013

Index

Introducció

En aquest document tècnic s'explicarà de forma senzilla el nostre projecte de crèdit de síntesis, que és crear un *Framework* per desenvolupar videojocs en 3D, escrit en C++11 i utilitzant l'API gràfica *Microsoft DirectX* 11.

La idea de crear un *Framework* és un projecte que sempre s'ha volgut fer no solament per la nostre passió per la industria, sinó que veiem un repte que podíem assolir i aprendre noves coses.

Durant la lectura del document s'ha volgut explicar tots els temes que passen per al lector quan pensa com i amb quines tecnologies s'ha de crear un *Framework*. Tota la informació ha sigut escrita per experiències que s'han tingut i investigacions fetes, i creiem que pot ajudar a més persones que vulguin crear el mateix projecte o un de semblant.

No s'ha entrat en temes especialment tècnics de la organització interna així com és *Microsoft DirectX* i C++ per que per això s'ha creat una documentació que explica cada classe i els seus mètodes. Es donà per suposat que el lector té un nivell mitja sobre el llenguatge de programació C++ i ha experimentat amb API's gràfiques.

És important tindre un nivell basic de matemàtiques, per tant s'ha inclòs un annex amb àlgebra bàsica.

Nota: Per una millor comprensió de Dementia és recomana consulta la documentació proporcionada.

Agraïments

Ens agradaria agrair als professors i companys aquets dos anys que hem cursat informàtica, que no solament ens han ajudat i ensenyat informàtica sinó també valors. Sobretot per l'ajuda i idees que s'han donat durant tot el desenvolupament del crèdit de síntesi, animar-nos a continuar un projecte quant era molt complex i no veiem possibilitats de aconseguir el nostre objectiu. Gracies.

Què és un *Framework*?

Entenem com a *Framework* o també anomenada biblioteca com un conjunt de fitxers amb un codi ja escrit que no es pot modificar i que serveix com a capa d'abstracció sobre una tecnologia, per què sigui més fàcil i productiu treballar amb ella. Amb aquesta premissa va sorgir el nostre projecte amb l'objectiu de crear una capa per sobre de la tecnologia *Microsoft DirectX 11*, i és quan va néixer *Dementia*.

Per més informació visita: http://es.wikipedia.org/wiki/*Framework*

Avui en dia existeix una gran varietat de *Frameworks* molt amplia, i n'hi ha des de codi obert fins a tancat, predominant més els oberts. A continuació inclourem els que creiem que són els més famosos. Tots són de codi lliure per què així es pugui observar com estan fets:

FITXA TÈCNICA

NOM	Ogre3D
DESENVOLUPADOR	Ogre3D Team
PROGRAMAT	C++
SISTEMA OPERATIU	Multiplataforma
LLICENCIA	MIT
ULTIMA VERSIÓ	1.9 RC1 (Unstable) 1.7.3 (Stable)
PAGINA WEB	http://www.ogre3d.org/

FITXA TÈCNICA

NOM	Irrlicht
DESENVOLUPADOR	Nikolaus Gebhardt
PROGRAMAT	C++
SISTEMA OPERATIU	Multiplataforma
LLICENCIA	Zlib
ULTIMA VERSIÓ	1.8 (Unstable) 1.7.2 (Stable)
PAGINA WEB	http://irrlicht.sourceforge.net/

FITXA TÈCNICA

NOM	JMonkey Engine
DESENVOLUPADOR	jME Team
PROGRAMAT	Java
SISTEMA OPERATIU	Multiplataforma
LLICENCIA	Licencia BSD
ULTIMA VERSIÓ	3.0 RC2 (Unstable) 2.1 (Stable)
PAGINA WEB	http://jmonkeyengine.com/

Codi lliure o codi privatiu?

Abans d'entrar en quin tipus és *Dementia*, podem explicar breument quina diferencia hi ha entre ells.

Els projectes o moviments amb format codi lliure és programari amb una idea molt clara, i és poder compartir el codi amb més gent, és a dir, el coneixement. Aquest tipus de moviment té molts beneficis sobre el projecte, no només per què hi participa més gent que si fos privatiu, sinó que normalment tendeix a tindre una qualitat superior que podria tindre un privatiu tècnicament.

No solament és per qüestions tècniques sinó també filosòfiques i morals, que són totalment oposades al programari propietari que és considerat des de el seu punt de vista anti ètic ja que no es comparteix el seu codi. Tot programari de codi obert té certs requisits que s'han de complir per considerar-se així mateix obert.

Per més informació sobre el codi obert: http://es.wikipedia.org/wiki/C%C3%B3digo_abierto

Per altra banda tenim el codi privatiu és on un projecte el seu codi no és difós entre la comunitat o altres persones. No és limita en l'accés de la font, sinó també les restriccions que s'hi puguin imposar, com limitacions d'ús, prohibició de modificació o redistribució. Al ser un programari amb drets d'autor l'usuari executarà el programa amb unes condicions que serà imposades.

Per més informació sobre el codi privatiu: http://es.wikipedia.org/wiki/Software_propietario

Un cop explicades les diferencies, argumentarem quina i per què l'hem triat. *Dementia* va néixer amb la intenció de ser un *Framework* de codi obert que pugui ajudar a més desenvolupadors a crear els seus propis videojocs en 3D, construir una comunitat activa i difondre el coneixement.

La idea a canviat des de l'inici, convertint-se en un projecte de codi tancat, el motiu d'aquest canvi que a principi no veiem, és que la inversió de recursos que s'ha fet i hi haurà en un futur és molt elevada, ja que crear una eina d'aquesta envergadura és complex. Per tant veiem lògic poder rentabilitzar els esforços invertits per poder seguir subsistint i crear nous projectes.

Un de les coses que ens hem fixat és que en la industria del videojoc el motor gràfic, o *Framework*, és quasi la carta més important que té una empresa a l'hora de elaborar el seus videojocs i desmarcar-se de la competència. Creiem que si volguéssim competir amb altres o sobreviure en aquest mercat tindríem que evitar que altres empreses poguessin utilitzar la mateixa tecnologia que nosaltres.

Finalment, els avantatges que ens aportava ser de codi obert eren moltes però en el tipus de societat i sistema econòmic actual ens és inviable, però no descartem que creem nous projectes enfocats de ser codi obert.

Introducció a la tecnologia *Microsoft DirectX* i C++

Durant la lectura apareix el nom de dos tecnologies molt importants i bases per aquest projecte, que són *Microsoft DirectX* i C++.

C++ és un llenguatge de programació que data dels inici dels 80 i pretén ser una millora al predecessor anomenat C. La principal diferencia entre aquests dos és que el primer incorpora la programació orientada a objectes, també coneguda com a POO, i és important mencionar que és compatible amb C ja que és una evolució d'aquest i podem escriure codi tant en C com C++.

Aquest llenguatge a sigut adoptat en grans projectes com sistemes operatius d'us actual Microsoft Windows, Mac OS X i Linux entre altres. Però el principal motiu per què hem triat C++ com a llenguatge de programació per crear *Dementia* ha sigut per què és un estàndard dintre de l'industria del videojoc, i per què hi ha moltes eines com llibreries de física que s'integren en projectes nomes escrits en aquest llenguatge.

També hi ha un altre motiu com és el alt rendiment que ofereix comparat amb altres competidors com Java, C# o Python, al ser un llenguatge compilat i no interpretat, ja que treballa a més baix nivell i només és compila un cop.

La tecnologia *Microsoft DirectX* és una API gràfica que proporciona la companyia de Redmon per crear aplicacions gràfiques per entorns Microsoft, ja sigui en el sistema operatiu Microsoft Windows, plataforma Xbox, tabletes Surface o Windows Phone.

Microsoft DirectX, com s'ha comentat és una API gràfica i per tant nomes són un conjunt de biblioteques que proporcionen unes funcions que treballen amb la gràfica a nivell assemblador, però nosaltres al interactuar amb la API diem que treballem a baix nivell per que toquem la gràfica constantment. Per utilitzar aquesta API s'ha de primer tindre instal·lades les llibreries i el SDK corresponent que proporciona Microsoft de forma gratuïta en la seva pagina.

La programació en DirectX es realitza o es recomanable fer-la amb C++, tot i que la pots fer en C sense cap problema, existeixen altre bindings com el de C# de forma no oficial tot i que en la pagina de Microsoft el recomana en cas que es vulgui treballar amb aquesta tecnologia.

Per què utilitzar *Microsoft DirectX* en comptes de OpenGL? Tot i que OpenGL sigui una alternativa lliure, la documentació és molt escassa i la dificultat és molt elevada comparada amb *Microsoft DirectX*, ja que aquest proporciona llibreries tant de matemàtiques, com de xarxa i de so, que OpenGL de base no. També és important que la diferencia de rendiment entre OpenGL i *Microsoft DirectX* és molt important, aquest últim ha demostrat ser molt més potent i més senzill d'aprendre.

Per finalitzar *Microsoft DirectX* és pot distribuir i obtenir també de forma gratuïta tot i que el seu codi és tancat.

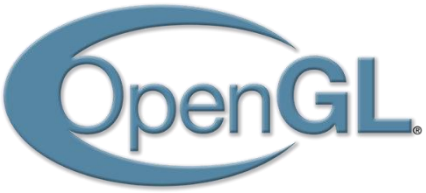
Per més informació sobre Microsoft DirectX: http://es.wikipedia.org/wiki/DirectX

Per més informació sobre OpenGL: http://es.wikipedia.org/wiki/OpenGL

Microsoft DirectX vs OpenGL

Microsoft DirectX i OpenGL mantenen un conflicte que va començar des de la aparició de *Microsoft DirectX* i que porta vigent. Les diferencies entre una i l'altre tècnicament són nul·les però la usabilitat és molt diferent, actualment les diferencies entre *Microsoft DirectX* 11 i OpenGL 4.0 cada cop són menors. La única diferencia entre ells és que Microsoft DirectX és de codi tancat al contrari de OpenGL.

Partint d'aquesta base sorgeixen les següents diferencies, i és que OpenGL al ser un estàndard de gràfics obert s'ha pogut realitzar un port a moltes més plataformes, i no solament funciona en pràcticament en el 100% de tots els dispositius existents (exceptuant Xbox i Microsoft Phone) , sinó que en molts sistemes operatius és la única API que tenen disponible.



Els sistemes operatiu que implementen OpenGL com a única API gràfica són Mac OS X i iOS de Apple, UNIX on deriven tots els Linux i Androids actuals i el sistema operatiu de Sony que incorpora Playstation. En el cas del sistema operatiu de la companyia de Redmon, Microsoft Windows, accepta les dues API per desenvolupar però té preferència per *Microsoft DirectX*, i en el cas de Xbox i Windows Phone actualment nomes suporten *Microsoft DirectX*.

Tot i que la lògica dona a entendre que el camí seria utilitzar OpenGL no és tan fàcil com pugui beures a primera vista per motius basics com, una documentació molt simple o pobre, no hi ha un entorn de desenvolupament i la segmentació de versions és molt elevada a diferencia del seu competidor. Però el punt més crític és que actualment és la quota dels sistemes operatius que nomes suporten OpenGL, és molt reduïda i no supera al 5% vers la 95% que té Microsoft Windows actualment.

És per això que moltes companyies tenen predilecció per *Microsoft DirectX,* ja que compleixen molt del cup de mercat, i tenen molt material ja creat amb aquesta tecnologia, realitzar un canvi podria suposar una inversió molt forta.

Actualment podríem dir que les coses estan canviant molt gracies al marcat de la telefonia mòbil, ja que nomes suporten OpenGL i això impulsa a molts desenvolupadors a utilitzar aquesta API gràfica.

La nostra elecció a sigut *Microsoft DirectX* per un motiu basic, per realitzar els primers passos amb una API gràfica és millor amb aquesta per totes les facilitats que donà, i la documentació és molt completa i això permet un desenvolupament ràpid, tot i que en un futur canviarem a OpenGL per que podrem arribar a més plataformes.

Què és *Dementia* i amb quin objectiu neix?

Dementia és un *Framework* escrit per ser orientat a objectes programat amb el nou estàndard de C++, el C++11, i utilitzem l'API gràfica que proporciona Microsoft, anomenada *Microsoft DirectX* 11, amb la que també està escrita en C++ per tant la interacció serà de 1:1.

El naixement de *Dementia* sorgeix per la necessitat no solament nostra, sinó del mercat sobre un *Framework* fàcil d'utilitzar, i que permeti desenvolupar videojocs en 3D amb un llenguatge de programació d'us professional com és el C++.

Es per això que veiem profitós crear aquest projecte, per què podem veure un nínxol de mercat que altres *Framework* no aconsegueixen omplir degut a que estan enfocats a estudis professionals i no independents com és el nostre cas.

Per tant es podria dir que *Dementia* intenta ocupar un sector del mercat per a petites companyies que vulguin crear videojocs amb un cost molt baix i amb el màxim benefici.

Quines plataformes té suport *Dementia*?

Dementia actualment nomes suporta la plataforma principal, el PC i sobre sistemes operatius Microsoft Windows. Pròximament serà ampliat a dos plataformes més com són la mòbil amb Windows Phone i la plataforma de consoles Xbox.

En un futur quan tinguem la tecnologia *Microsoft DirectX* finalitzada i implementada crearem la implementació per l'API gràfica OpenGL al Framework degut a que suporta l'altre meitat de mercat que també volem accedir-hi, sobretot en els mòbils tant de Apple com de Google que nomes suporten OpenGL ES 1.0+.

Descartem totalment centrar-nos en plataformes de consola com són la Playstation, Nintendo Wii,Nintendo WiiU i les corresponents portàtils PSP Vita i Nintendo DS/3DS.

Quin sector en el mercat ocuparà *Dementia* i aquí va dirigit?

El mercat de *Framework*s per desenvolupar aplicacions en 3D no està molt estes i hi ha molt per explotar i cobrir. Actualment la gran majoria de projectes o de motors gràfics estan pensats per desenvolupadores o grans equips amb un pressupost molt elevat i no per a grups independents. Aquí és on *Dementia* entra en joc, per cobrir aquesta secció de mercat, tot i que la nostre idea com a empresa no es la de desenvolupar *Framework*s sinó la de videojocs i tenim que crear-lo igualment, pot suposar un extra de beneficis per finançar els nostres projectes.

La intenció de *Dementia* és ser un *Framework* de gran velocitat en execució i una elevada producció amb el qual puguis crear les teves aplicacions en 3D amb molt poc cost econòmic i treure el màxim benefici possible. També suposa una alternativa més econòmica que les possibilitats actuals que hi ha en el mercat.

L'usuari mitja d'aquest *Framework* és el d'un programador experimentat o amb coneixements suficients en matèria gràfica i de llenguatge de programació com és el cas de C++.

Quina motivació donà fer *Dementia*

Dementia sorgeix com el projecte més complex que hem realitzat fins a data d'avui. Tenim tres problemes molt grans, el primer de tots és que tenim un coneixement baix de programació gràfica per tant tindrem que aprendre a programar en *Microsoft DirectX* i saber també com s'hi programa en Win32 per poder interaccionar amb el sistema operatiu.

Un cop superat aquest obstacle existeix un més complex, que és aprendre C++11, tot i que sapiguem C i tenim un coneixement baix de C++, utilitzarem moltes funcionalitats avançades per poder crear el *Framework*.

Finalment i com a últim obstacle suposarà crear el *Framework* en si, mai hem creat un i no hi ha una guia que puguis saber com començar ja que és quelcom que no es pot preveure ni planificar com un estàndard, ja que cadascú programa el *Framework* segons les seves necessitats, comptem amb un coneixement nul en aquest aspecte.

Tot i que el temps que comptem per fer-ho és d'un total d'un mes partint de zero des de el primer moment, suposa un repte i un risc que estem disposats a córrer. Creiem que qualsevol altre projecte més fàcil no hagués suposat un repte, tot i que el tindríem acabat abans, no haguéssim après tant.

Quins objectius té el grup amb *Dementia*?

L'objectiu principal d'aquest projecte exceptuant l'esmentat anteriorment seria una cosa tant bàsica com mostrar objectes per pantalla. A primera vista pot semblar una cosa bastant trivial, ja que sinó tindria sentit la creació del *Framework,* però és molt més complex ja que tens que preparar la pipeline de la gràfica per les ordres del usuari.

Un cop la pipeline estigui preparada desenvolupar les demes coses com llums, textures, etc... serà molt més fàcil per que es muntar sobre una base solida. Per tant considerem un èxit assolir que *Dementia* poguí mostrar objectes per pantalla. Com objectius extres, seria implementar textures i llums sobre els objectes.

Eines utilitzades

Per crear *Dementia* s'han utilitzat relativament poques eines, ja que treballem bastant a baix nivell i nomes necessitem un editor per desenvolupar i la API gràfica que ens proporciona Microsoft.

Com a entorn de desenvolupament hem triat el Microsoft Visual Studio Premium 2012, com era evident, és l'únic IDE que té la documentació actualitzada i proporciona suficients ajudes per poder escriure DirectX.

Microsoft Visual Studio Premium 2012 és una eina molt potent i utilitzada per entorn professionals i per desenvolupar macro projectes, però el principal punt fort que té, és la possibilitat de obtenir tota la informació de una compilació en Debug de que esta passant per el codi, des de la inicialització de la variable fins el que escriu en Assembler poder optimitzar al màxim el rendiment. Finalment també té un Debug de fitxers HSLS també anomenats shaders que serà molt útil quan tinguin problemes de compilació.

La següent eina que ens ha ajudat molt per mantenir un històric de versions, és el Microsoft Team Foundation 2012, que s'integra amb el IDE Microsoft Visual Studio Premium 2012 i ens permet realitzar copies de seguretat i mantenir un control de versions. Incorpora moltes eines útils com un gestor d'errors que un dels integrants pot pujar un informe d'error explicant que com s'ha produït, on i quins passos s'han fet per què es dones el error.

L'últim programa utilitzat és el *Microsoft DirectX* SDK, que incorpora exemples i ajudes que seran necessàries quan estiguem treballant amb el *Framework*. Aquest programa no conté cap aparença gràfica ja que nomes son les llibreries per poder desenvolupar *Microsoft DirectX*.

Podria suposar una gran inversió de diners utilitzar tot aquest programari, però el Institut posa a disposició llicencies Dreamspark que podem obtenir les llicencies de forma gratuïta al ser estudiants.

Per més informació de Microsoft Team Foundation visita: http://msdn.microsoft.com/es-es/vstudio/ff637362.aspx

Per més informació de Microsoft Visual Studio 2012: http://www.microsoft.com/visualstudio/esn

Metodologia de treball

Una metodologia de treball és una procés en que tots els components del grup tenen un rol, i consisteix en tindre un control total sobre el projecte que s'està desenvolupant.

Per mes informació met. desenvolupament visita: http://es.wikipedia.org/wiki/Metodolog%C3%ADa_de_desarrollo_de_software

Nosaltres hem aplicat una metodologia que portem temps utilitzant i és la metodologia de la **pair programming** o també coneguda com a programació en parelles. Pot semblar poc eficient que dos programadors treballin en un mateix lloc, en que un programa i l'altre observa, però és sense dubte la metodologia més eficient en cas que dos programadors siguin compatibles entre ells dos.

En el nostre cas al tindre molta experiència programant entre nosaltres aquesta metodologia ens permet programar projectes molt grans en poc temps i amb una possibilitat d'error molt reduïda.

Per més informació de pair programming visita: http://es.wikipedia.org/wiki/Programaci%C3%B3n_en_pareja

Característiques de *Dementia*

Les característiques de *Dementia* en les seves primeres versions inicials proporciona unes característiques bàsiques que permeten crear objectes, associar-los a nodes, crear llums i aplicar textures.

La idea és mantenir sempre que es pugui la màxima simplicitat i la forma amb que s'hi interactua amb ell. Les característiques que suporta actualment són:

- ***Lectura de dispositius:***
  - *Dementia* posa a disposició un gestor de devices (dispositius) que permetrà al programador saber quina tecla o pulsació de ratolí ha realitzat mitjançant els mètodes basics d'interacció que proporcionen els teclat i ratolí.
  - Els dispositius que suporta actualment son:
    - Teclat
    - Ratolí
- ***Camara en primera persona***
  - *Dementia* incorpora el que creiem una de les càmeres més utilitzades en videojocs, i és la càmera en primera persona, que ens permetrà simular un moviment de vista realista, no confondre amb una càmera lliure. L'usuari podrà configurar la velocitat de moviment així com l'angle de visió, a traves de les coordenades del ratolí.
- ***Textures***
  - *Dementia* incorpora la carrega de textures bàsica on podràs a cada element assignar-li una textura.
- ***Sistema de Nodes***
  - *Dementia* incorpora un sistema de nodes on cada element esta subjecte a un node, per així poder manipular els objectes d'un escenari sense grans esforços. Aquest tipus de sistema serà comentant en profunditat en un apartat dedicat per ell.
- ***Instancing***
  - *Dementia* incorpora la aquesta tecnologia on tots els objectes del mateix tipus són creats un únic cop però comparteixen la mateixa referencia per així poder tindre un numero indeterminat de duplicats sense sacrificar rendiment.
- ***Geometry Factory***
  - *Dementia* incorpora una fabrica de figures geomètriques on podràs crear qualsevol figura de les que té disponibles amb una única línia.
- ***Llums***
  - *Dementia* incorpora un sistema de llums bàsiques on tenim els tres tipus, una direccional, una de posició i finalment una de focus.

Prerequisits necessaris

Existeixen uns prerequisits per poder entendre el contingut del *Dementia* en la seva totalitat, considerem que per entendre una part bàsica podria ser suficient amb una base de C i C++ i son totes aquelles classes que son una abstracció sobre *Microsoft DirectX*.

La gran majoria de les classes són necessaris uns coneixements elevats de matemàtiques així com Àlgebra i geometria avançada, experiència en C++11, com treballa el hardware en aquest cas la CPU/Memòria i Gràfica (la seva memòria i GPU també estan inclòs) i saber programar en *Microsoft DirectX*.

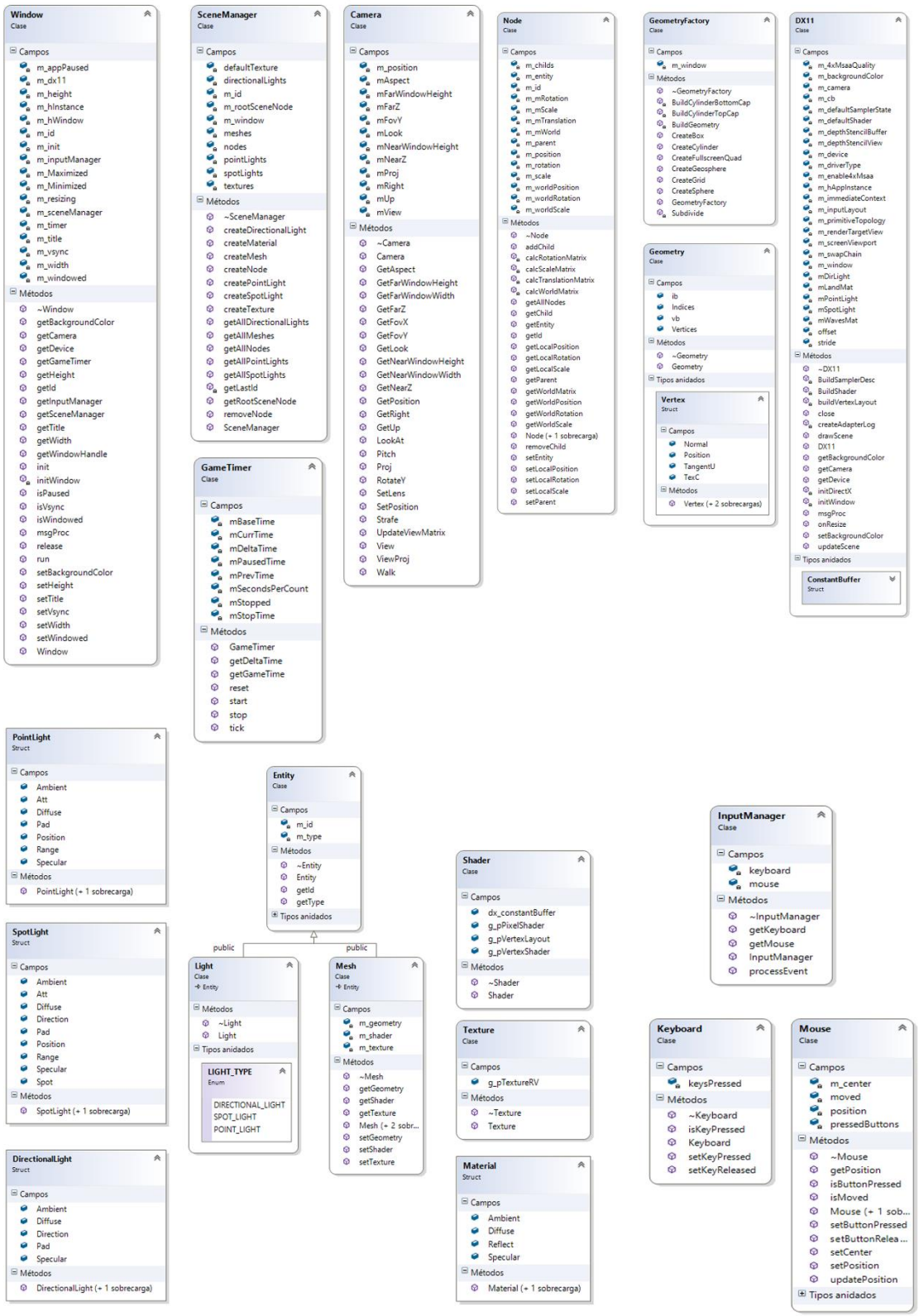
S'incorporarà un annex amb informació que es cregui necessària per poder entendre parts de *Dementia*.

Les entranyes de *Dementia*

Dementia tot i ser un projecte relativament petit comparat amb altres *Framework*s i de poques classes conté molta informació i s'ha optimitzat al màxim per obtenir un rendiment excel·lent. La idea de com s'ha estructurat és que volem separar el màxim possible de *Microsoft DirectX* a l'usuari, per què sinó la idea de *Framework* perd totalment el seu sentit.

***Nota**: No s'entrarà en detalls en profund de com està estructurada la classe, sinó quina és la seva funció bàsica. Creiem més oportú en cas de voler més informació consultar la documentació adjuntada.*

***Nota II:** Durant l'explicació d'una classe apareixen referencies a altres classes, en cas de dubte o millor coneixement sobre aquesta, la documentació proporciona la informació necessària.*


Window
Clase
Campos
m_appPaused
m_dx11
m_height
m_hInstance
m_hWindow
m_id
m_init
m_inputManager
m_Maximized
m_Minimized
m_resizing
m_sceneManager
m_timer
m_title
m_vsync
m_width
m_windowed
Métodos
~Window
getBackgroundColor
getCamera
getDevice
getGameTimer
getHeight
getId
getInputManager
getSceneManager
getTitle
getWidth
getWindowHandle
init
initWindow
isPaused
isVsync
isWindowed
msgProc
release
run
setBackgroundColor
setHeight
setTitle
setVsync
setWidth
setWindowed
Window
SceneManager
Clase
Campos
defaultTexture
directionalLights
m_id
m_rootSceneNode
m_window
meshes
nodes
pointLights
spotLights
textures
Métodos
~SceneManager
createDirectionalLight
createMaterial
createMesh
createNode
createPointLight
createSpotLight
createTexture
getAllDirectionalLights
getAllMeshes
getAllNodes
getAllPointLights
getAllSpotLights
getLastId
getRootSceneNode
removeNode
SceneManager
Camera
Clase
Campos
m_position
mAspect
mFarWindowHeight
mFarZ
mFovY
mLook
mNearWindowHeight
mNearZ
mProj
mRight
mUp
mView
Métodos
~Camera
Camera
GetAspect
GetFarWindowHeight
GetFarWindowWidth
GetFarZ
GetFovX
GetFovY
GetLook
GetNearWindowHeight
GetNearWindowWidth
GetNearZ
GetPosition
GetRight
GetUp
LookAt
Pitch
Proj
RotateY
SetLens
SetPosition
Strafe
UpdateViewMatrix
View
ViewProj
Walk
Node
Clase
Campos
m_childs
m_entity
m_id
m_mRotation
m_mScale
m_mTranslation
m_mWorld
m_parent
m_position
m_rotation
m_scale
m_worldPosition
m_worldRotation
m_worldScale
Métodos
~Node
addChild
calcRotationMatrix
calcScaleMatrix
calcTranslationMatrix
calcWorldMatrix
getAllNodes
getChild
getEntity
getId
getLocalPosition
getLocalRotation
getLocalScale
getParent
getWorldMatrix
getWorldPosition
getWorldRotation
getWorldScale
Node (+ 1 sobrecarga)
removeChild
setEntity
setLocalPosition
setLocalRotation
setLocalScale
setParent
GeometryFactory
Clase
Campos
m_window
Métodos
~GeometryFactory
BuildCylinderBottomCap
BuildCylinderTopCap
BuildGeometry
CreateBox
CreateCylinder
CreateFullscreenQuad
CreateGeosphere
CreateGrid
CreateSphere
GeometryFactory
Subdivide
DX11
Clase
Campos
m_4xMsaaQuality
m_backgroundColor
m_camera
m_cb
m_defaultSamplerState
m_defaultShader
m_depthStencilBuffer
m_depthStencilView
m_device
m_driverType
m_enable4xMsaa
m_hAppInstance
m_immediateContext
m_inputLayout
m_primitiveTopology
m_renderTargetView
m_screenViewport
m_swapChain
m_window
mDirLight
mLandMat
mPointLight
mSpotLight
mWavesMat
offset
stride
Métodos
~DX11
BuildSamplerDesc
BuildShader
buildVertexLayout
close
createAdapterLog
drawScene
DX11
getBackgroundColor
getCamera
getDevice
initDirectX
initWindow
msgProc
onResize
setBackgroundColor
updateScene
Tipos anidados
ConstantBuffer
Struct
Geometry
Clase
Campos
ib
Indices
vb
Vertices
Métodos
~Geometry
Geometry
Tipos anidados
Vertex
Struct
Campos
Normal
Position
TangentU
TexC
Métodos
Vertex (+ 2 sobrecargas)
GameTimer
Clase
Campos
mBaseTime
mCurrTime
mDeltaTime
mPausedTime
mPrevTime
mSecondsPerCount
mStopped
mStopTime
Métodos
GameTimer
getDeltaTime
getGameTime
reset
start
stop
tick
PointLight
Struct
Campos
Ambient
Att
Diffuse
Pad
Position
Range
Specular
Métodos
PointLight (+ 1 sobrecarga)
SpotLight
Struct
Campos
Ambient
Att
Diffuse
Direction
Pad
Position
Range
Specular
Spot
Métodos
SpotLight (+ 1 sobrecarga)
DirectionalLight
Struct
Campos
Ambient
Diffuse
Direction
Pad
Specular
Métodos
DirectionalLight (+ 1 sobrecarga)
Entity
Clase
Campos
m_id
m_type
Métodos
~Entity
Entity
getId
getType
Tipos anidados
public
public
Light
Clase
Entity
Métodos
~Light
Light
Tipos anidados
LIGHT_TYPE
Enum
DIRECTIONAL_LIGHT
SPOT_LIGHT
POINT_LIGHT
Mesh
Clase
Entity
Campos
m_geometry
m_shader
m_texture
Métodos
~Mesh
getGeometry
getShader
getTexture
Mesh (+ 2 sobr...
setGeometry
setShader
setTexture
Shader
Clase
Campos
dx_constantBuffer
g_pPixelShader
g_pVertexLayout
g_pVertexShader
Métodos
~Shader
Shader
Texture
Clase
Campos
g_pTextureRV
Métodos
~Texture
Texture
Material
Struct
Campos
Ambient
Diffuse
Reflect
Specular
Métodos
Material (+ 1 sobrecarga)
InputManager
Clase
Campos
keyboard
mouse
Métodos
~InputManager
getKeyboard
getMouse
InputManager
processEvent
Keyboard
Clase
Campos
keysPressed
Métodos
~Keyboard
isKeyPressed
Keyboard
setKeyPressed
setKeyReleased
Mouse
Clase
Campos
m_center
moved
position
pressedButtons
Métodos
~Mouse
getPosition
isButtonPressed
isMoved
Mouse (+ 1 sob...
setButtonPressed
setButtonRelea...
setCenter
setPosition
updatePosition
Tipos anidados

Dementia::Window

La classe *Dementia*::Window és la responsable de gestionar tota la finestra, principalment dels esdeveniments que sorgeixen com poden ser les senyals d'entrada i sortida.

Dementia::Window té un constructor que serveix per inicialitzar els paràmetres de com serà la finestra però encara no està creada, ja que per això tindrem que cridar el seu mètode Init on s'inicialitzarà i s'hi guardarà referencies a objectes de les classes que necessitin aquesta finestra i també s'iniciarà a si mateixa.

S'encarrega de la gestió de esdeveniments d'entrada i sortida que són redirigits cap a una classe anomenada *Dementia*::InputManager quan són senyals de tipus de E/S. Les demes senyals que no pertanyin aquest tipus són tractades per la finestra, com poden ser les següents:

- ***WM_ACTIVATE***: Si la finestra no esta activa el temporitzador de *Dementia*::GameTimer és pausat.
- ***WM_SIZE***: Si la finestra és redimensionada guardarem les noves dimensions en les corresponents variables. Després es poden gestionar nous esdeveniments com son:
    - ***SIZE_MINIMIZED***:
        - Sabrem si la aplicació a sigut minimitzada, per tant direm que la aplicació esta pausada i pararà el temporitzador de *Dementia*::GameTimer.
    - ***SIZE_MAXIMIZED***:
        - Sabrem si la aplicació a sigut maximitzada, per tant direm que la aplicació esta en execució i reactivarà el temporitzador de *Dementia*::GameTimer.
    - ***SIZE_RESTORED***:
        - Sabrem segons els booleans que s'han activat en els dos anteriors mètodes (Minimized i Maximized) si *Dementia*::DX11 té que actualitzar el seu ViewPort.
- ***WM_ENTERSIZEMOVE***
    - Aquest tipus d'esdeveniment saltarà quan l'usuari redimensioni la finestra, es on aturarem el temporitzador de *Dementia*::GameTimer.
- ***WM_EXITSIZEMOVE***
    - Aquest tipus d'esdeveniment saltarà quan l'usuari hagi finalitzat la redimensionament de la finestra i reactivarà el temporitzador de *Dementia*::GameTimer.
- ***WM_DESTROY***
    - Aquest tipus d'esdeveniment saltarà quan la finestra s'hagi de destruir.
- ***WM_MENUCHAR***
    - Aquest tipus d'esdeveniment saltarà quan realitzem un Alt + Enter per entrar en mode finestra completa no realitzi un beep.
- ***WM_GETMINMAXINFO***
    - Aquest tipus d'esdeveniment saltarà quan vulguem redimensionar una finestra i especifiquem un mínim de dimensions que pot tindre la finestra en el nostre cas és de 200 x 200 i impedeixi ser inferior aquestes dimensions.

Per finalitzar l'explicació de la classe *Dementia*::Window, conté variables privades que emmagatzemen referencies a objectes d'altres classes que són consultades a traves d'altres

punts, com poden ser la referencia al objecte *Dementia*::InputManager i *Dementia*::DX11. Per últim s'encarrega d'enviar senyals de parada i resumir el temps a la classe *Dementia*::GameTimer.

Dementia::GameTimer

La classe *Dementia*::GameTimer s'encarrega de gestionar el temps que s'executa en l'aplicació és a dir, quan el programa s'executa s'inicia amb temps zero i va contant el temps transcorregut fins la seva finalització.

Per ser mes precisos, rep senyals per part de la finestra quan aquesta és minimitzada o rep algun canvi de dimensions que indiquin que aturi el temps per no comptar temps morts.

La idea de tindre una classe que porti el temps d'execució és molt important en cas que vulguem sincronitzar moviments d'un personatge, vulguem que alguns botons s'activin transcorregut un temps, etc... També és important per *Dementia*::DX11 per a l'hora de realitzar un refresc de pantalla, ja que s'encarrega de mesurar les imatges per segon que produeix la aplicació.

Dementia::Keyboard

La classe *Dementia*::Keyboard realitza funcions molt bàsiques de teclat, necessàries per la interacció entre el programa i l'usuari. Les seves principals característiques són que podem saber si una tecla és premuda, i podem dir que premi una tecla concreta. El total de tecles suportades ve donat per el sistema operatiu, en aquest Microsoft Windows.

Dementia::Mouse

La classe *Dementia*::Mouse realitza funcions molt bàsiques de ratolí necessàries per la interacció entre el programa i l'usuari. Les seves principals característiques són que podem saber si un boto del ratolí és premuda, i podem especificar una posició, així com obtenir-la. El total de botons del ratolí suportada son el estàndard actual, esquerra, dreta, els botons laterals 4 i 5, i la roda quan realitza de boto.

Dementia::Entity

La classe *Dementia*::Entity és una classe per encapsular altres classes com són *Dementia*::Mesh. La seva creació és necessària per què té un atribut que totes les classes que heretin d'ella tindran i és un identificador necessari per poder eliminar o buscar. Així si en un futur volguéssim que un node pogués tindre més coses associades que un Mesh com podria ser una llum podríem tractar-los tots per igual.

Dementia::Light

La classe *Dementia*::Light és una classe també encapsuladora que permet crear diferents tipus de llums sense la necessitat de tractar-les per separat, com poden ser la spot light, directional light i point light.

Dementia::Camera

La classe *Dementia*::Camera és una classe que s'encarrega d'una part complexa amb la interacció amb l'usuari i és poder navegar per el escenari de forma lliure, però en aquest cas es tracta de una càmera en primera persona.

Dementia::Geometry

La classe *Dementia*::Geometry és una classe que encapsula les figures en forma de estructura, on emmagatzema els index i vèrtex de la figura, els seus corresponents buffers que serveixen per què la gràfica pugui renderitzar la figura per pantalla.

Dementia::GeoemtryFactory

La classe *Dementia*::GeometryFactory és una classe que s'encarrega de crear les figures geomètriques a traves d'uns dissenys predefinits però que l'usuari pot canviar la figura a traves dels paràmetres que rep. La funció principal per el que es va crear és per proporcionar al usuari una manera còmode de crear figures primitives de forma molt senzilla.

Dementia::InputManager

La classe *Dementia*::InputManager és una classe que s'encarrega de gestionar tots els possibles i actuals dispositius que suporta el *Framework*, des de rebre els esdeveniment que són enviats per part de la finestra fins a saber quin esdeveniment enviar en cada dispositiu específic.

Dementia::MathHelper

La classe *Dementia*::MathHelper és una classe que té com a funció ser una llibreria matemàtica on s'han escrit formules genèriques que utilitzarem constantment en el *Framework*, és complement amb la llibreria matemàtica que proporciona *Microsoft DirectX*.

Dementia::Mesh

La classe *Dementia*::Mesh és una classe contenidor que l'usuari utilitzarà per vincular-lo a un node, i és en aquesta classe on guardem la geometria, la corresponent textura i el shader.

Dementia::Node

La classe *Dementia*::Node és una classe que permet al usuari manipular tots els objectes que veiem per pantalla amb facilitat, és a dir, un node té associat un element de la pantalla. Cada node pot tindre infinits fills i així successivament, i a traves de l'herència els fills obtenen les posicions, rotacions i escales dels pares i son aplicades conjuntament amb les de fill fen així un arbre de nodes.

Dementia::SceneManager

La classe *Dementia*::ScenenManager és una classe on l'usuari interactuarà amb el motor quan vulguem afegir qualsevol element, hi ha que és qui controla quins objectes són creats i quins es destrueixen. Aquesta classe emmagatzema vectors de cada tipus dels elements que poden existir en el *Framework*, que seran necessaris a l'hora de renderitzar un frame.

Dementia::Shader

La classe *Dementia*::Shader és una classe on s'hi guarda el que es coneix com a constant buffer que és una estructura idèntica a la que podem trobar en el shader, i que ens permet transportar dades des de el *Framework* cap al fitxer shader.

Dementia::Texture

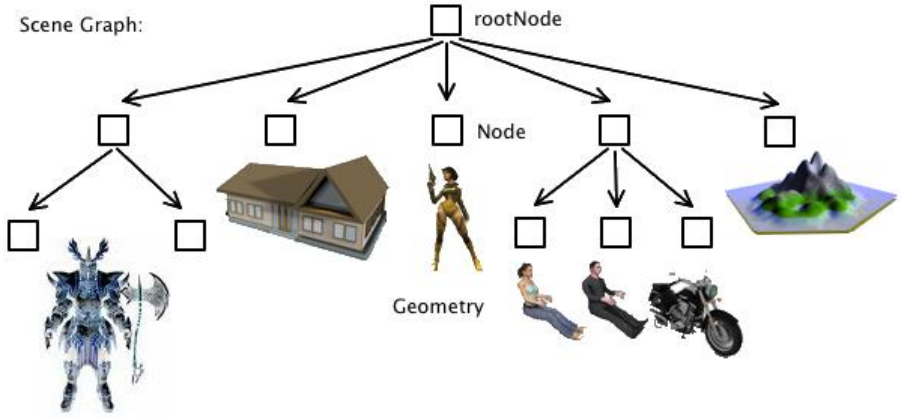
La classe *Dementia*::Texture és una classe on s'hi gestiona les textures que podem aplicar als nostres objectes.

Dementia::DX11

La classe *Dementia*::DX11 és una classe on s'encarrega de tot el procés de renderitzar tots els objectes que hi hagi en memòria, crear els respectius buffers i preparar tots els elements per existeixi una comunicació entre l'usuari i la gràfica.

Sistema de nodes

El sistema de nodes o també conegut com Scene Graph és la part més important que té el *Framework*, ja que s'encarrega de que tots els objectes que nosaltres incloem en la finestra estan lligats a nodes, que en el seu conjunt formen un arbre d'herència.

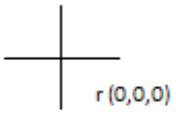


Tot comença amb un node principal o arrel que el crea automàticament *Dementia* amb el nom de RootSceneNode. Aquest node és el pare de tots els futurs fills que hi puguin penjar d'ell. Un node nomes pot tindre un objecte de tipus *Dementia*::Entity que pot ser una llum o un objecte geomètric, però cada node pot tindre indefinits fills convertint-se en un node pare.

Els nodes tenen tres atributs essencials que son: la seva posició, rotació i escala. Cada node al ser fill d'un altre obté els atributs del pare i afegeix els nous que incorpora el fill, és a dir, que si un node el tenim en la posició X i aquest té un fill que hi penja en la posició Y, el fill en realitat esta en la posició X + Y, això és diu posició en el món global.

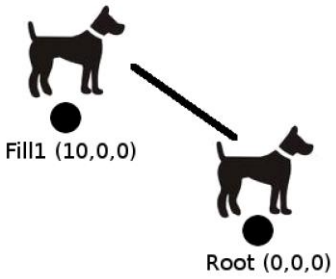
Per tant un node pot estar en un món global i en un local, les principals diferencies és que nosaltres treballem en local quan modifiquem un sol node, i en global quan aquest s'afegeix a la resta de nodes. El principal motiu de perquè s'utilitza aquest procés és per que en programació gràfica és més còmode tindre un objecte posicionat en el centre i treballar amb referencies locals que no pas amb globals que poden estar en constant canvi.

Per entrar en més profunditat de com funciona aquest sistema explicarem una successió de casos. Comencem amb un node al centre, que serà el Root.

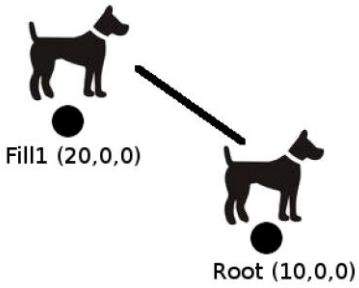


Afegrim aquest node un objecte de tipus *Dementia*::Entity, en cas una figura geomètrica en forma canina.

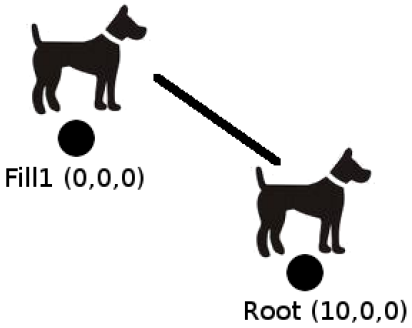
Actualment el Root és troba en el centre de tot el mon, i afegirem un nou node amb un altre gos com a objecte i que sigui fill de Root. El nou node es trobarà en la posició (10,0,0).



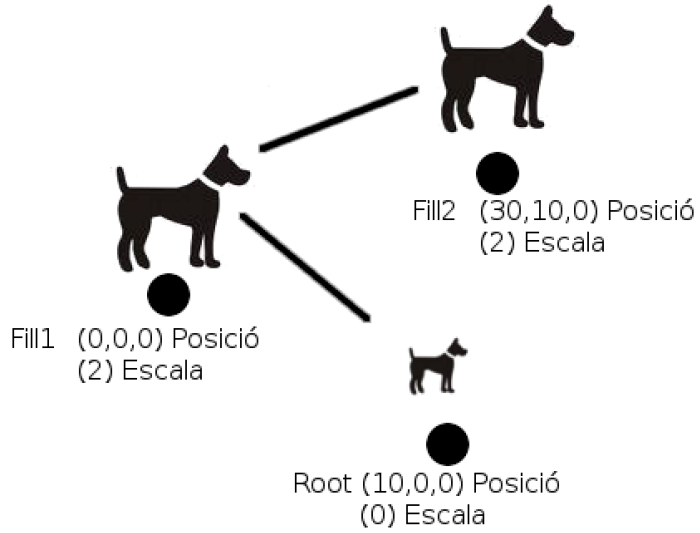
Si ara realitzéssim una operació de translació al node pare (Root) a la següent posició (10,0,0) el resultat és que el Fill1 incrementa la seva posició en (20,0,0).



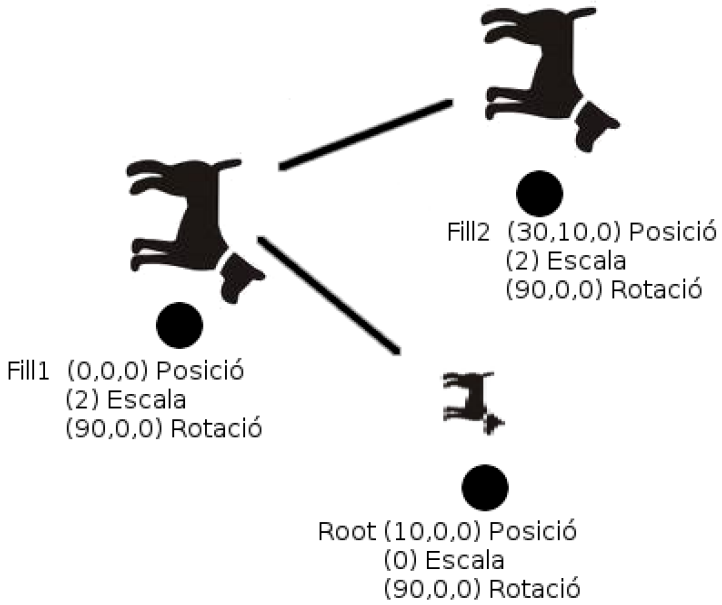
El motiu d'aquest canvi sobre el fill, és que existeix una herència i si el node pare canvia de posició a una nova, com a resultat afecta a tots els fills. Si el nou fill tingues una posició (-10,0,0) la conseqüència és que estaria posicionat al centre del escena.



Afegirem un nou fill, anomenat Fill2 a la posició (30,10,0), i que serà fill de Fill1, per tant Fill1 serà pare, i aquest l'escalarem dos vegades les seves dimensions.



El resultat és que el Root al no tindre cap escala comparat amb els altres nodes és de ser un objecte més petit, però com el pare de Fill2 té una escala, aquesta es heretada a Fill2. Si ara girem el node Root 90º el resultat seria el següent.



El sistema de nodes és una tècnica que té molts avantatges per l'usuari encara que a principi pugui semblar més problemàtica, ja que no manipules un node específic sinó un tots els descendents d'aquest. Les millores que aporta són poder moure molts nodes o realitza escena amb una facilitat que tractat els nodes per separat no es podria o seria molt més complexa. Per poder utilitzar aquesta tècnica en el *Framework* crearem els nodes en el *Dementia*::SceneManager i ell automàticament ens gestiona tots aquests processos.

Curiositats durant el procés de Crèdit de Síntesis

Durant el procés de crèdit de síntesis hi hagut moltes curiositats unes més importants i altres que passen mes desapercebudes, així que destacarem les que creiem que son les més imprescindibles.

Com a curiositat que s'emporta casi el primer lloc és que *Dementia* ha tingut tres versions, és a dir, que hem creat de zero *Dementia* tres cops. El que podria semblar a primera vista una bogeria en un projecte de crèdit de síntesis pel poc temps que tenim, ha sigut la millor estratègia.

La primera versió partíem d'un esquema de *Framework* de propòsit general, però no s'adaptava a les nostres necessitats ja que qual volíem ampliar funcionalitats, o poder fer que l'usuari pogués afegir més objectes no donava de si el esquema.

La segona versió corregia tot aquest problema, però sorgia un de pitjor, és que tot i haver avançat encara ni en la primera versió ni la segona no mostrava res per la finestra, es a dir, la pipeline de la gràfica no s'omplia per una mala implementació del core de *Dementia*.

Això va suposar una gran frustració per nosaltres i es va decidir crear un de nou corregint tots els errors de les versions anteriors, un molt millor on la separació entre *Microsoft DirectX* i el usuari fos casi total. La idea va funcionar i la implementació i creació nomes es va trigar entre dos a tres dies entre pensar com seria i programar-ho. Aquesta nova versió és la actual i que segurament apareixerà la quarta versió on millorarem molts aspectes que hem anat veient en aquesta, com una millor gestió de API gràfiques independentment quina s'utilitzi sigui OpenGL o *Microsoft DirectX*.

La següent curiositat i que creiem que és molt important, és que durant tres dies sencers a l'hora de implementar la característica de llums hem tingut problemes i és que no es transferien les estructures entre la memòria del processador a la memòria de la gràfica.

Això va portar molts problemes, per què no sabíem perquè unes dades si es transferien i d'altres no. Es va provar tots els mètodes, posant dades per defecte i tot tipus de idees que podien funcionar però cap acabava de funcionar.

El problema és va solucionar a tres dies de la data de entrega quan donàvem la característica de llums com perduda, i resulta que durant la creació de la nova versió del *Framework*, la 2.0, va quedar una estructura en forma de residu la qual fèiem un sizeof i les dimensions d'aquesta amb el que utilitzàvem era molt més petita. Quan vam descobrir que el problema era això va ser una alegria per què per fi tindríem llums, però també va ser molt frustrant que per un error de eliminar un residu dones tants problemes.

Per últim i no menys important hi ha curiositats menors però hi ha una i és que durant el desenvolupament de *Dementia* hem comparat les nostres estadístiques de rendiment amb altres i el nostre possiblement en la majoria de casos te un rendiment superior del 50% respecte l'altre.

Conclusions globals sobre *Dementia*

Les conclusions són global ja que el projecte ha sigut desenvolupat entre els dos un al costat de l'altre i per tant la opinió és quasi idèntica per tant aquestes són les nostres conclusions.

Les nostres conclusions sobre el projecte *Dementia* han sigut més que satisfactòries, no nomes perquè hem pogut fer un crèdit de síntesis que sempre hem volgut, sinó que ha sigut un èxit total i s'han complit i assolit tots els objectius proposats.

Actualment podríem dir que crear *Dementia* és una experiència que estaríem disposats a repetir per seguir millorant com a programadors, ja que podríem dir que és una de les coses més complexes per sota de crear el teu propi sistema operatiu (algun dia pot ser que ens en fem un).

Ara ja tenim un projecte de futur com a equip i persones i és millorar l'eina que hem creat independentment del benefici econòmic que ens pugui aportar, sinó per el prestigi que pugui donar. També actuarà com a targeta de presentació cap a altres companyies que s'hi dediquin a programar videojocs en 3D i poder oferir una alternativa als nostres competidors, una rapida, fàcil d'utilitzar i econòmica.

Possiblement cap projecte ens ha donat tantes alegries i desgracies per parts iguals, quan una funcionalitat funciona és una avanç molt important i t'anima a seguir, però per aconseguir que aquest funcioni primer s'ha de sofrir molt per aconseguir-ho i molts cops desesperes i no t'agradaria continuar amb el projecte. Aquest tipus de sensacions són molt habituals en projectes d'aquestes característiques on hi ha molts factors en contra, on l'ordre dels factors canvia totalment el producte.

Quan va començar el crèdit de síntesis, ja teníem experiència de com crear projectes i sabíem que primer millor era documentar-se el màxim possible i preparar un bon diagrama de classes i pensar com fer-ho tot encara que sacrifiquéssim temps de desenvolupament. Gracies aquest pensament durant el desenvolupament ha sigut molt ràpid i en qüestió de dies s'ha pogut implementar tot el que volíem fer i ampliar amb nous objectius ha sigut molt fàcil gracies a la bona estructuració.

Durant el procés de desenvolupament considerem que s'han creat dos versions de *Dementia*, la primera va sorgir a la segona setmana de crèdit però a finals d'aquesta no funcionava cap cosa correctament, i la idea de *Framework* s'havia perdut totalment ja que el usuari escrivia *Microsoft DirectX*. Aquella setmana va ser bastant desagradable i inclús pensàvem abandonar el projecte perquè no veiem temps ni ànims suficients per continuar, fins que per part de professorat es va proposar que agaféssim un descans i no deixéssim el projecte. Possiblement va ser lo millor que hem fet en aquest projecte, no solament va servir per desconnectar sinó que a una setmana del final del projecte hem redissenyat tot *Dementia* amb el que considerem la versió 2.0, i la implementació va ser molt mes rapida que l'anterior. Això es degut a que hem après dels anteriors errors de la primera versió, teníem ganes de continuar i no deixar-ho, i exceptuant algun problema puntual ha funcionat tot o casi tot a la primera.

Per finalitzar donar gracies a totes aquelles persones que ens a donat el suport durant el procés de crèdit síntesis.

Annex

L'annex conté matemàtiques en angles per què la millor documentació que existeix sobre aquests temes està en aquest idioma. Creiem que una traducció perjudicaria els conceptes.

1. Àlgebra bàsica

1.1 Vectors

A *vector* refers to a quantity that possesses both magnitude and direction. Quantities that possess both magnitude and direction are called *vectorvalued quantities*. Examples of vector-valued quantities are forces (a force is applied in a particular direction with a certain strength — magnitude), displacements (the net direction and distance a particle moved), and velocities (speed and direction). Thus, vectors are used to represent forces, displacements, and velocities. We also use vectors to specify pure directions, such as the direction the player is looking in a 3D game, the direction a polygon is facing, the direction in which a ray of light travels, or the direction in which a ray of light reflects off a surface.

A first step in characterizing a vector mathematically is geometrically: We graphically specify a vector by a directed line segment (see Figure 1.1), where the length denotes the magnitude of the vector and the aim denotes the direction of the vector. We note that the location in which we draw a vector is immaterial because changing the location does not change the magnitude or direction (the two properties a vector possesses). That is, we say two vectors are equal if and only if they have the same length and they point in the same direction. Thus, the vectors **u** and **v** drawn in Figure 1.1a are actually equal because they have the same length and point in the same direction. In fact, because location is unimportant for vectors, we can always translate a vector without changing its meaning (since a translation changes neither length nor direction). Observe that we could translate **u** such that it completely overlaps with **v** (and conversely), thereby making them indistinguishable — hence their equality. As a physical example, the vectors **u** and **v** in Figure 1.1b both tell the ants at two different points, *A* and *B*, to move north ten meters from their current location. Again we have that **u = v**. The vectors themselves are independent of position; they simply instruct the ants how to move from where they are. In this example, they tell the ants to move north (direction) ten meters (length).

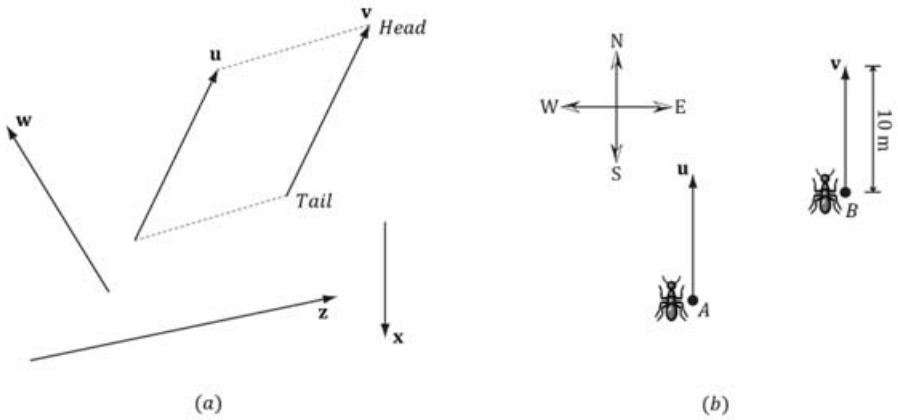


Figure 1.1: (a) Vectors drawn on a 2D plane. (b) Vectors instructing ants to move ten meters north.

1.2 Vectors and Coordinate Systems

We could now define useful geometric operations on vectors, which can then be used to solve problems involving vector-valued quantities. However, since the computer cannot work with vectors geometrically, we need to find a way of specifying vectors numerically instead. So what we do is introduce a 3D coordinate system in space, and translate all the vectors so that their tails coincide with the origin (Figure 1.2). Then we can identify a vector by specifying the coordinates of its head, and write **v** = (*x*, *y*, *z*) as shown in Figure 1.3. Now we can represent a vector with three floats in a computer program.

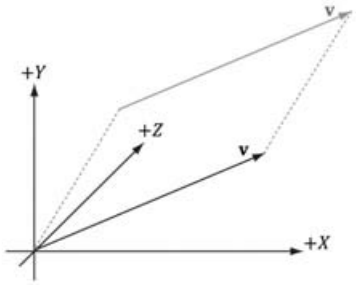


Figure 1.2: We translate **v** so that its tail coincides with the origin of the coordinate system. When a vector's tail coincides with the origin, we say that it is in *standard position*.

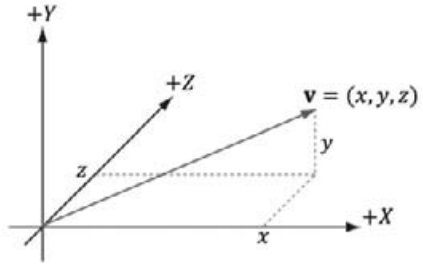


Figure 1.3: A vector specified by coordinates relative to a coordinate system.

Consider Figure 1.4, which shows a vector **v** and two frames in space.We can translate **v** so that it is in standard position in either of the two frames. Observe, however, that the coordinates of the vector **v** relative to frame *A* are different from the coordinates of the vector **v** relative to frame *B*. In other words, the *same* vector has a different coordinate representation for distinct frames.

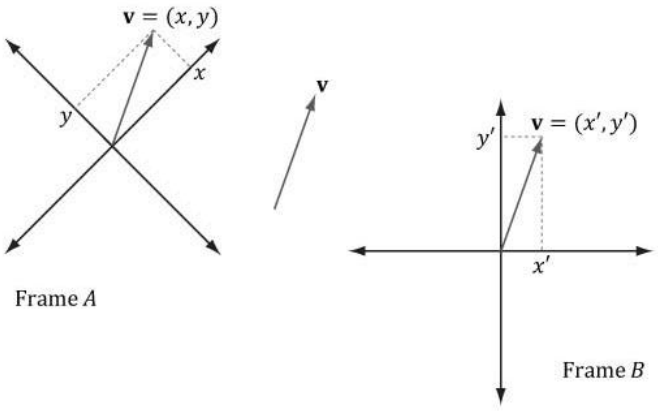


Figure 1.4: The *same* vector **v** has different coordinates when described relative to different frames.

The idea is analogous to, say, temperature. Water boils at 100º Celsius or 212º Fahrenheit. The physical temperature of boiling water is the *same* no matter the scale (i.e., we can't lower the boiling point by picking a different scale), but we assign a different scalar number to the temperature based on the scale we use. Similarly, for a vector, its direction and magnitude, which are embedded in the directed line segment, does not change; only the coordinates of it change based on the frame of reference we use to describe it. This is important because it means whenever we identify a vector by coordinates, those coordinates are relative to some frame of reference. Often in 3D computer graphics we will utilize more than one frame of reference and, therefore, will need to keep track of which frame the coordinates of a vector are described relative to; additionally, we will need to know how to convert vector coordinates from one frame to another.

1.3 Basic Vector Operations

We now define equality, addition, scalar multiplication, and subtraction on vectors using the coordinate representation.

- Two vectors are equal if and only if their corresponding components are equal. Let $\mathbf{u} = (u_x u_y u_z)$, and $\mathbf{v} = (v_x v_y v_z)$. Then $\mathbf{u} = \mathbf{v}$ if and only if $u_x = v_x$, $u_y = v_y$, and $u_z = v_z$.
- We add vectors componentwise; as such, it only makes sense to add vectors of the same dimension. Let $\mathbf{u} = (u_x u_y u_z)$, and $\mathbf{v} = (v_x v_y v_z)$. Then $\mathbf{u} + \mathbf{v} = (u_x + v_x, u_y + v_y, u_z + v_z)$.
- We can multiply a scalar (i.e., a real number) and a vector, and the result is a vector. Let k be a scalar, and let $\mathbf{u} = (u_x u_y u_z)$, then $k\mathbf{u} = (ku_x, ku_y, ku_z)$. This is called scalar multiplication.
- We define subtraction in terms of vector addition and scalar multiplication. That is, $\mathbf{u} + (-\mathbf{v}) = (u_x - v_x, u_y - v_y, u_z - v_z)$.

1.4 Length and Unit Vectors

Geometrically, the magnitude of a vector is the length of the directed line segment. We denote the magnitude of a vector by double vertical bars (e.g. $\|\mathbf{u}\|$ denotes the magnitude of $\mathbf{u}$). Now, given a vector $\mathbf{v} = (v_x v_y v_z)$, we wish to compute its magnitude algebraically. The magnitude of a 3D vector can be computed by applying the Pythagorean Theorem twice; see Figure 1.8.

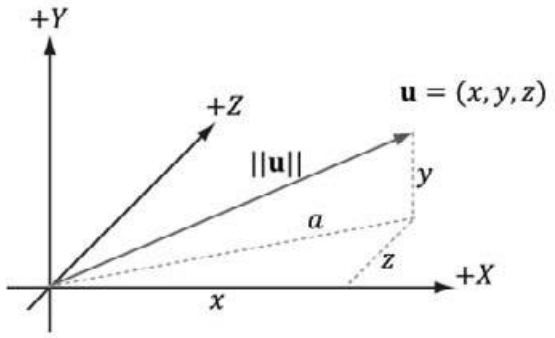


Figure 1.8: The 3D length of a vector can be computed by applying the Pythagorean theorem twice.

First, we look at the triangle in the xz-plane with sides x, z, and hypotenuse a. From the Pythagorean theorem, we have $\sqrt{x^2 + y^2}$. Now look at the triangle with sides a, y, and hypotenuse $\|\mathbf{u}\|$. From the Pythagorean theorem again, we arrive at the following magnitude formula:

$$\|\mathbf{u}\| = \sqrt{y^2 + a^2} = \sqrt{y^2 + \left(\sqrt{x^2 + z^2}\right)^2} = \sqrt{x^2 + y^2 + z^2} \quad \text{(Eq 1.1)}$$

For some applications, we do not care about the length of a vector because we want to use the vector to represent a pure direction. For such directiononly vectors, we want the length of the vector to be exactly one. When we make a vector unit length, we say that we are *normalizing* the vector. We can normalize a vector by dividing each of its components by its magnitude:

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|} = \left(\frac{x}{\|\mathbf{u}\|}, \frac{y}{\|\mathbf{u}\|}, \frac{z}{\|\mathbf{u}\|}\right) \quad \text{(Eq 1.2)}$$

To verify that this formula is correct, we can compute the length of $\hat{\mathbf{u}}$:

$$\|\hat{\mathbf{u}}\| = \sqrt{\left(\frac{x}{\|\mathbf{u}\|}\right)^2 + \left(\frac{y}{\|\mathbf{u}\|}\right)^2 + \left(\frac{z}{\|\mathbf{u}\|}\right)^2} = \frac{\sqrt{x^2 + y^2 + z^2}}{\sqrt{\|\mathbf{u}\|^2}} = \frac{\|\mathbf{u}\|}{\|\mathbf{u}\|} = 1$$

So $\hat{\mathbf{u}}$ is indeed a unit vector.

1.4 The Dot Product

The dot product is a form of vector multiplication that results in a scalar value; for this reason, it is sometimes referred to as the scalar product. Let $\mathbf{u} = (u_x u_y u_z)$, and $\mathbf{v} = (v_x v_y v_z)$; then the dot product is defined as follows:

$$\mathbf{u} \cdot \mathbf{v} = (u_x + v_x, u_y + v_y, u_z + v_z).$$

In words, the dot product is the sum of the products of the corresponding components. The dot product definition does not present an obvious geometric meaning. Using the law of cosines, we can find the relationship,

$$u * v = ||u|| \, ||v|| \cos\theta \text{ (Eq 1.4)}$$

where θ is the angle between the vectors **u** and **v** such that $0 \leq \theta \leq \pi$. So, Equation 1.4 says that the dot product between two vectors is the cosine of the angle between them scaled by the vectors' magnitudes. In particular, if both **u** and **v** are unit vectors, then $\mathbf{u} \cdot \mathbf{v}$ is the cosine of the angle between them (i.e., $\mathbf{u} \cdot \mathbf{v} = \cos\theta$).

Equation 1.4 provides us with some useful geometric properties of the dot product:

- If $\mathbf{u} \cdot \mathbf{v} = 0$, then $\mathbf{u} \perp \mathbf{v}$ (i.e., the vectors are orthogonal).
- If $\mathbf{u} \cdot \mathbf{v} > 0$, then the angle θ between the two vectors is less than 90° (i.e., the vectors make an acute angle).
- If $\mathbf{u} \cdot \mathbf{v} < 0$, then the angle θ between the two vectors is greater than 90° (i.e., the vectors make an obtuse angle).

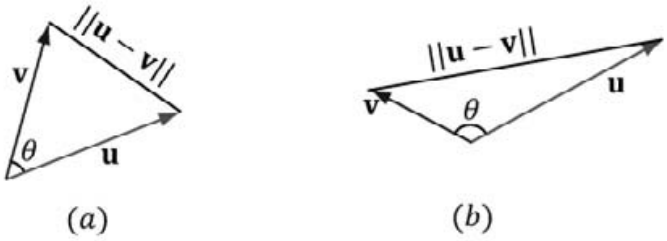
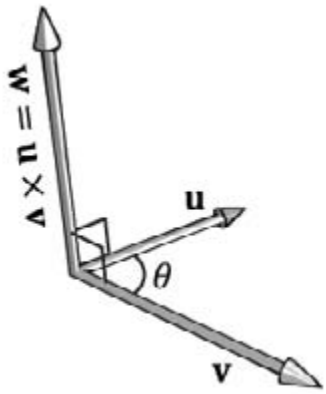


Figure 1.9: On the left, the angle θ between **u** and **v** is an acute angle. On the right, the angle θ between **u** and **v** is an obtuse angle. When we refer to the angle between two vectors, we always mean the smallest angle, that is, the angle θ such that $0 \leq \theta \leq \pi$.



1.5 The Cross Product

The second form of multiplication vector math defines is the cross product. Unlike the dot product, which evaluates to a scalar, the cross product evaluates to another vector; moreover, the cross product is only defined for 3D vectors (in particular, there is no 2D cross product). Taking the cross product of two 3D vectors **u** and **v** yields another vector, **w** that is mutually orthogonal to **u** and **v**. By that we mean **w** is orthogonal to **u**, and **w** is orthogonal to **v** (see Figure 1.11). If $\mathbf{u}=(u_x, u_y, u_z)$ and $\mathbf{v}=(v_x, v_y, v_z)$, then the cross product is computedlike so:

$$w = u \times v = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x) \text{ (Eq 1.5)}$$

Figure 1.11: The cross product of two 3D vectors, **u** and **v**, yields another vector, **w**, that is mutually orthogonal to **u** and **v**. If you take your left hand and aim the fingers in the direction of the first vector **u**, and then curl your fingers toward **v** along an angle $0 \leq \theta \leq \pi$, then your thumb roughly points in the direction of $\mathbf{w} = \mathbf{u} \times \mathbf{v}$; this is called the *left-handthumb rule*.

1.6 Points

So far we have been discussing vectors, which do not describe positions. However, we will also need to specify positions in our 3D programs; for example, the position of 3D geometry and the position of the 3D virtual camera. Relative to a coordinate system, we can use a vector in standard position (see Figure 1.12) to represent a 3D position in space; we call this a *position vector*. In this case, the location of the tip of the vector is the characteristic of interest, not the direction or magnitude. We will use the terms "position vector" and "point" interchangeably since a position vector is enough to identify a point.

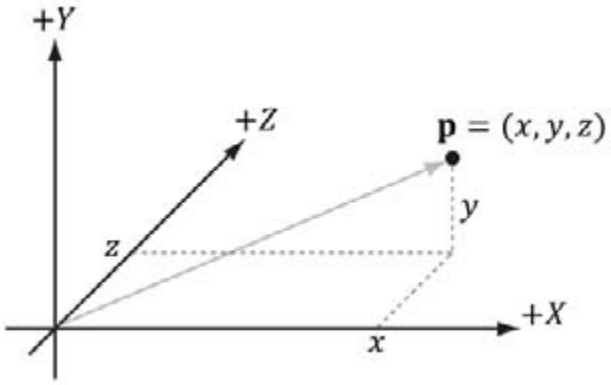


Figure 1.12: The position vector, which extends from the origin to the point, fully describes the location of the point relative to the coordinate system.

One side effect of using vectors to represent points, especially in code, is that we can do vector operations that do not make sense for points; for instance, geometrically, what should the sum of two points mean? On the other hand, some operations can be extended to points. For example, we define the difference of two points **q** – **p** to be the vector from **p** to **q**. Also, we define a point **p** plus a vector **v** to be the point **q** obtained by displacing **p** by the vector **v**. Conveniently, because we are using vectors to represent points relative to a coordinate system, no extra work needs to be done for the point operations just discussed as the vector algebra *Framework* already takes care of them (see Figure 1.13).

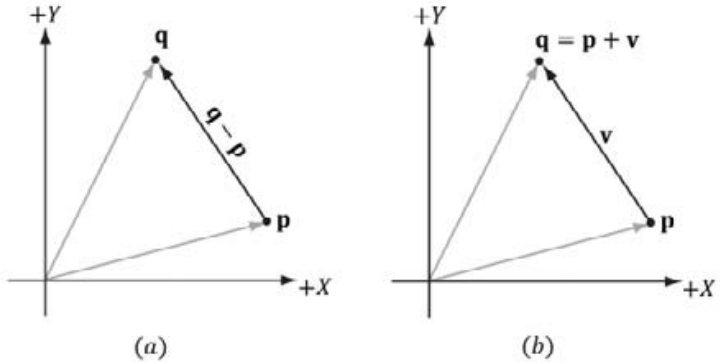


Figure 1.13:
(a) The difference **q** – **p** between two points is defined as the vector from **p** to **q**. (b) A point **p** plus the vector **v** is defined to be the point **q** obtained by displacing **p** by the vector **v**.

2. Matrix Algebra

2.1 Definition

An *m x n matrix* **M** is a rectangular array of real numbers with *m* rows and *n* columns. The product of the number of rows and columns gives the dimensions of the matrix. The numbers in a matrix are called *elements* or *entries*. We identify a matrix element by specifying the row and column of the element using a double subscript notation M_{ij}, where the first subscript identifies the row and the second subscript identifies the column.

Consider the following matrices:

$$\mathbf{A}=\begin{bmatrix} 3.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 2 & -5 & \sqrt{2} & 1 \end{bmatrix} \quad \mathbf{B}=\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} \quad \mathbf{u}=\begin{bmatrix} u_1, u_2, u_3 \end{bmatrix} \quad \mathbf{v}=\begin{bmatrix} 1 \\ 2 \\ \sqrt{3} \\ \pi \end{bmatrix}$$

- Matrix **A** is a 4_4 matrix; matrix **B** is a 3 × 2 matrix; matrix **u** is a 1 × 3 matrix; and matrix **v** is a 4 × 1 matrix.
- We identify the element in the fourth row and second column of matrix **A** by A_{42}=–5. We identify the element in the second row and first column of matrix **B** by B_{21}.
- Matrices **u** and **v** are special matrices in the sense that they contain a single row or column, respectively. We sometimes call these kinds of matrices *row vectors* or *column vectors* because they are used to represent a vector in matrix form (e.g., we can freely interchange the vector notations (*x*, *y*, *z*) and [*x*, *y*, *z*]). Observe that for row and column vectors, it is unnecessary to use a double subscript to denote the elements of the matrix — we only need one subscript.

Occasionally we like to think of the rows of a matrix as vectors. For example, we might write:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} \leftarrow \mathbf{A}_{1,*} \rightarrow \\ \leftarrow \mathbf{A}_{2,*} \rightarrow \\ \leftarrow \mathbf{A}_{3,*} \rightarrow \end{bmatrix}$$

where $\mathbf{A}_{1,*} = [A_{11}, A_{12}, A_{13}]$, $\mathbf{A}_{2,*} = [A_{21}, A_{22}, A_{23}]$, and $\mathbf{A}_{3,*} = [A_{31}, A_{32}, A_{33}]$. In this notation, the first index specifies the row, and we put an asterisk (*) in the second index to indicate that we are referring to the entire row vector. Likewise, we can do the same thing for the columns:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ \mathbf{A}_{*,1} & \mathbf{A}_{*,2} & \mathbf{A}_{*,3} \\ \downarrow & \downarrow & \downarrow \end{bmatrix}$$

Where

$$\mathbf{A}_{*,1}=\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix}, \quad \mathbf{A}_{*,2}=\begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix}, \text{ and } \mathbf{A}_{*,3}=\begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix}$$

In this notation, the second index specifies the column, and we put an asterisk (*) in the first index to indicate that we are referring to the entire column vector. We now define equality, addition, scalar multiplication, and subtraction on matrices:

- Two matrices are equal if and only if their corresponding elements are equal; as such, two matrices must have the same number of rows and columns in order to be compared.
- We add two matrices by adding their corresponding elements; as such, it only makes sense to add matrices that have the same number of rows and columns.
- We multiply a scalar and a matrix by multiplying the scalar with every element in the matrix.
- We define subtraction in terms of matrix addition and scalar multiplication. That is, $\mathbf{A} - \mathbf{B} = \mathbf{A} + (-1 \cdot \mathbf{B}) = \mathbf{A} + (-\mathbf{B})$.

Let

$$\mathbf{A}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}, \mathbf{B}=\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}, \mathbf{C}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}, \text{and } \mathbf{D}=\begin{bmatrix}2 & 1 & -3\\ -6 & 3 & 0\end{bmatrix}$$

Then,

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}+\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1+6 & 5+2\\ -2+5 & 3+(-8)\end{bmatrix}=\begin{bmatrix}7 & 7\\ 3 & -5\end{bmatrix}$$

$$\mathbf{A}=\mathbf{C}$$

$$3\mathbf{D}=3\begin{bmatrix}2 & 1 & -3\\ -6 & 3 & 0\end{bmatrix}=\begin{bmatrix}3(2) & 3(1) & 3(-3)\\ 3(-6) & 3(3) & 3(0)\end{bmatrix}=\begin{bmatrix}6 & 3 & -9\\ -18 & 9 & 0\end{bmatrix}$$

$$\mathbf{A}-\mathbf{B}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}-\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1-6 & 5-2\\ -2-5 & 3-(-8)\end{bmatrix}=\begin{bmatrix}-5 & 3\\ -7 & 11\end{bmatrix}$$

2.2 Matrix Multiplication

The next section defines how to multiply two matrices together.

2.2.1 Definition

If **A** is an $m \times n$ matrix and **B** is an $n \times p$ matrix, then the product **AB** is defined and is an $m \times p$ matrix **C**, where the ijth entry of the product **C** is given by taking the dot product of the ith row vector in **A** with the jth column vector in **B**, that is,

$$C_{ij} = \mathbf{A}_{i,*} \cdot \mathbf{B}_{*,j} \quad (EQ\ 2.1)$$

So note that in order for the matrix product **AB** to be defined, we require that the number of columns in **A** be equal to the number of rows in **B**, which is to say, we require that the dimension of the row vectors in **A** equal the dimension of the column vectors in **B**. If these dimensions did not match, then the dot product in Equation 2.1 would not make sense.

Let

$$\mathbf{A}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix} \text{ and } \mathbf{B}=\begin{bmatrix}2 & -6\\ 1 & 3\\ -3 & 0\end{bmatrix}$$

The product **AB** is not defined since the row vectors in **A** have a dimension of 2 and the column vectors in **B** have a dimension of 3. In particular, we cannot take the dot product of the first row vector in **A** with the first column vector in **B** because we cannot take the dot product of a 2D vector with a 3D vector.

Let

$$\mathbf{A}=\begin{bmatrix}-1 & 5 & -4\\ 3 & 2 & 1\end{bmatrix} \text{ and } \mathbf{B}=\begin{bmatrix}2 & 1 & 0\\ 0 & -2 & 1\\ -1 & 2 & 3\end{bmatrix}$$

We first point out that the product **AB** is defined (and is a 2×3 matrix) because the number of columns of **A** equals the number of rows of **B**. Applying Equation 2.1 yields:

$$\begin{aligned}\mathbf{AB}&=\begin{bmatrix}-1 & 5 & -4\\ 3 & 2 & 1\end{bmatrix}\begin{bmatrix}2 & 1 & 0\\ 0 & -2 & 1\\ -1 & 2 & 3\end{bmatrix}\\ &=\begin{bmatrix}(-1,5,-4)\cdot(2,0,-1) & (-1,5,-4)\cdot(1,-2,2) & (-1,5,-4)\cdot(0,1,3)\\ (3,2,1)\cdot(2,0,-1) & (3,2,1)\cdot(1,-2,2) & (3,2,1)\cdot(0,1,3)\end{bmatrix}\\ &=\begin{bmatrix}2 & -19 & -7\\ 5 & 1 & 5\end{bmatrix}\end{aligned}$$

Observe that the product **BA** is not defined because the number of columns in **B** does *not* equal the number of rows in **A**. This demonstrates that, in general, matrix multiplication is not commutative; that is, **AB** ≠ **BA**.

2.2.2 Vector-Matrix Multiplication

Consider the following vector-matrix multiplication:

$$\mathbf{uA}=[x,y,z]\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{22} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=[x,y,z]\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ \mathbf{A}_{*,1} & \mathbf{A}_{*,2} & \mathbf{A}_{*,3}\\ \downarrow & \downarrow & \downarrow\end{bmatrix}$$

Observe that **uA** evaluates to a 1_3 row vector in this case. Now, applying Equation 2.1 gives:

$$\begin{aligned}\mathbf{uA} &= \begin{bmatrix}\mathbf{u}\cdot\mathbf{A}_{*,1}, & \mathbf{u}\cdot\mathbf{A}_{*,2}, & \mathbf{u}\cdot\mathbf{A}_{*,3}\end{bmatrix} \\ &= \begin{bmatrix}xA_{11}+yA_{21}+zA_{31}, & xA_{12}+yA_{22}+zA_{32}, & xA_{13}+yA_{23}+zA_{33}\end{bmatrix} \\ &= \begin{bmatrix}xA_{11}, xA_{12}, xA_{13}\end{bmatrix}+\begin{bmatrix}yA_{21}, yA_{22}, yA_{23}\end{bmatrix}+\begin{bmatrix}zA_{31}, zA_{32}, zA_{33}\end{bmatrix} \\ &= x\begin{bmatrix}A_{11}, A_{12}, A_{13}\end{bmatrix}+y\begin{bmatrix}A_{21}, A_{22}, A_{23}\end{bmatrix}+z\begin{bmatrix}A_{31}, A_{32}, A_{33}\end{bmatrix} \\ &= x\mathbf{A}_{1,*}+y\mathbf{A}_{2,*}+z\mathbf{A}_{3,*}\end{aligned}$$

Thus,

$$uA = xA_{1,*} + zA_{3,*} \text{ (Eq 2.2)}$$

Equation 2.2 is an example of a *linear combination*, and it says that the vector-matrix product **uA** is equivalent to a linear combination of the row vectors of the matrix **A** with scalar coefficients *x*, *y*, and *z* given by the vector **u**. Note that, although we show this for a 1_3 row vector and a 3_3 matrix, the result is true in general. That is, for a 1×*n* row vector **u** and an *n*×*m* matrix **A**, we have that **uA** is a linear combination of the row vectors in **A** with scalar coefficients given by **u**:

$$\begin{bmatrix}u_1,\ldots,u_n\end{bmatrix}\begin{bmatrix}A_{11} & \cdots & A_{1m}\\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm}\end{bmatrix} = u_1\mathbf{A}_{1,*}+\cdots+u_n\mathbf{A}_{n,*} \quad \text{(Eq 2.3)}$$

2.2.3 Associativity

Matrix multiplication has some nice algebraic properties. For example, matrix multiplication distributes over addition: **A**(**B** + **C**) = **AB** + **AC**. In particular, however, we will use the associative law of matrix multiplication from time to time, which allows us to choose the order in which we multiply matrices:

$$(AB)C = A(BC)$$

2.3 The Transpose of a Matrix

The *transpose* of a matrix is found by interchanging the rows and columns of the matrix. Thus the transpose of an *m x n* matrix is an *n x m* matrix. We denote the transpose of a matrix **M** as $\mathbf{M}_T$. Find the transpose for the following three matrices:

$$\mathbf{A}=\begin{bmatrix}2 & -1 & 8\\ 3 & 6 & -4\end{bmatrix} \quad \mathbf{B}=\begin{bmatrix}a & b & c\\ d & e & f\\ g & h & i\end{bmatrix} \quad \mathbf{C}=\begin{bmatrix}1\\ 2\\ 3\\ 4\end{bmatrix}$$

The transposes are found by interchanging the rows and columns, thus

$$\mathbf{A}^T = \begin{bmatrix} 2 & 3 \\ -1 & 6 \\ 8 & -4 \end{bmatrix} \quad \mathbf{B}^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \quad \mathbf{C}^T = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

2.4 The Indentity Matrix

There is a special matrix called the *identity matrix*. The identity matrix is a square matrix that has zeros for all elements except along the main diagonal; the elements along the main diagonal are all ones. For example, below are 2 x 2, 3 x 3, and 4 x 4 identity matrices.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The identity matrix acts as a multiplicative identity; that is, if **A** is an *m* x *n* matrix, **B** is an *n* x *p* matrix, and **I** is the *n* x *n* identity matrix, then

$$AI = A \text{ and } IB = B$$

In other words, multiplying a matrix by the identity matrix does not change the matrix. The identity matrix can be thought of as the number 1 for matrices. In particular, if **M** is a square matrix, then multiplication with the identity matrix is commutative:

$$MI = IM = M$$

2.5 Inverse of a Matrix

Matrix algebra does not define a division operation, but it does define a multiplicative inverse operation. The following list summarizes the important information about inverses:

- Only square matrices have inverses; therefore, when we speak of matrix inverses we assume we are dealing with a square matrix.
- The inverse of an *n x n* matrix **M** is an *n x n* matrix denoted as $\mathbf{M}^{-1}$.
- Not every square matrix has an inverse. A matrix that does have an inverse is said to be *invertible*, and a matrix that does not have an inverse is said to be *singular*.
- The inverse is unique when it exists.
- Multiplying a matrix with its inverse results in the identity matrix: $\mathbf{M}\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$. Note that multiplying a matrix with its own inverse is a case when matrix multiplication is commutative.

Matrix inverses are useful for solving for other matrices in a matrix equation. For example, suppose that we are given the matrix equation $\mathbf{p}^T = \mathbf{pM}$. Further suppose that we are given $\mathbf{p}^T$ and **M**, and want to solve for **p.** Assuming that **M** is invertible (i.e., $\mathbf{M}^{-1}$ exists), we can solve for **p** like so:

$p' = pM$	
$p'M^{-1} = pMM^{-1}$	Multiply both sides of the equation by $\mathbf{M}^{-1}$.
$p'M^{-1} = pI$	$\mathbf{MM}^{-1} = \mathbf{I}$, by definition of inverse.
$p'M^{-1} = p$	$\mathbf{pI} = \mathbf{p}$, by definition of the identity matrix.

Techniques for finding inverses are beyond the scope of this book, but they are described in any linear algebra textbook (it is not difficult; it is just not worth digressing into the procedure here). In §2.6 we will learn about a D3DX function that will find the inverse of a matrix for us, and in the next chapter we will simply give the inverses of the important types of matrices that we will work with in this book. To conclude this section on inverses, we present the following useful algebraic property for the inverse of a product:

$$(AB)^{-1} = B^{-1} A^{-1}$$

This property assumes both **A** and **B** are invertible and that they are both square matrices of the same dimension. To prove that $\mathbf{B}^{-1}\mathbf{A}^{-1}$ is the inverse of **AB**, we must show $(\mathbf{AB})(\mathbf{B}^{-1}\mathbf{A}^{-1}) = \mathbf{I}$ and $(\mathbf{B}^{-1}\mathbf{A}^{-1})(\mathbf{AB}) = \mathbf{I}$. This is done as follows:

$$(AB)\,(B^{-1}A^{-1}) = A(BB^{-1})A^{-1} = AIA^{-1} = AA^{-1} = I$$
$$(B^{-1}A^{-1})(AB) = B^{-1}(A^{-1}A)B = B^{-1}IB = B^{-1}B = I$$

Bibliografia

Documentació oficial de *Microsoft DirectX*: http://msdn.microsoft.com/

Documentació no oficial de *Microsoft DirectX*: Introduction to 3D Game Programming with DirectX 11 – Frank D. Luna (http://www.d3dcoder.net/)

Forums de consulta especialitzats en desenvolupament de vidoejocs: http://www.gamedev.net/page/index.html

Forum de consulta de programació http://stackoverflow.com/

Tutorials de directX provieids per *Microsoft DirectX* SDK Samples