LỜI CẢM ƠN

Trước hết em xin chân thành thầy Lê Thụy là giáo viên hướng dẫn em trong quá tình thực tập. Thầy đã giúp em rất nhiều và đã cung cấp cho em nhiều tài liệu quan trọng phục vụ cho quá trình tìm hiểu về đề tài "Tìm hiểu về Lập trình đồ họa trên Symbian".

Thứ hai, Em xin chân thành cảm ơn các thầy cô trong bộ môn công nghệ thông tin đã chỉ bảo bảo em trong quá trình học và rèn luyện trong 4 năm học vừa qua. Đồng thời em cảm ơn các bạn sinh viên lớp CT901 đã gắn bó với em trong quá trình rèn luyện tại trường.

Cuối cùng em xin chân thành cảm ơn ban giám hiệu trường Đại Học Dân Lập Hải Phòng đã tạo điều kiện cho em có kiến thức, thư viện của trường là nơi mà sinh viên trong trường có thể thu thập tài liệu trợ giúp cho bài giảng trên lớp. Đồng thời các thầy cô trong trường giảng dạy cho sinh viên kinh nghiệm cuộc sống. Với kiến thức và kinh nghiệm đó sẽ giúp em cho công việc và cuộc sống sau này.

Em xin chân thành cảm ơn!

Hải Phòng, tháng 7 năm 2009.

Sinh viên

Phan Mạnh Cường

MỤC LỤC

Mở đầu

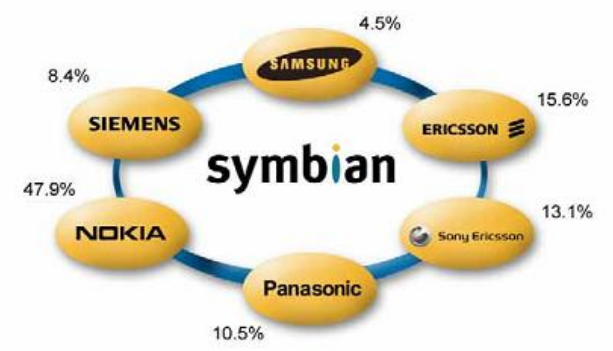
Hiện nay ngành công nghiệp phần mềm đang rất phát triển và ngành công nghiệp phần mềm trên điện thoại di động cũng không nằm ngoài xu thế đó. Tuy còn nhiều hạn chế trong phần cứng của điện thoại di động nhưng nó đã thể hiện được sức mạnh của mình trong rất nhiều các lĩnh vực khác nhau như giải trí, các tiện ích, thanh toán điện tử v.v… Ngành công nghệ phần mềm di động vẫn còn khá non trẻ ở Việt Nam vì vậy việc nghiên cứu và phát triển các ứng dụng trên di động là rất cần thiết.

Em nhận thấy nhu cầu của con người trong lĩnh vực giải trí trên di động ngày càng cao. Chính vị vậy em thực hiên đề tài này nhằm hiểu rõ về các kĩ thuật lập trình trên thiết bị động, đặc biệt là các kĩ thuật xây dựng đồ họa 3D trên di động để có thể tạo nên một game 3D hoàn chỉnh

CHƯƠNG 1: Tổng quan về hệ điều hành Symbian và điện thoại thông minh Series 60

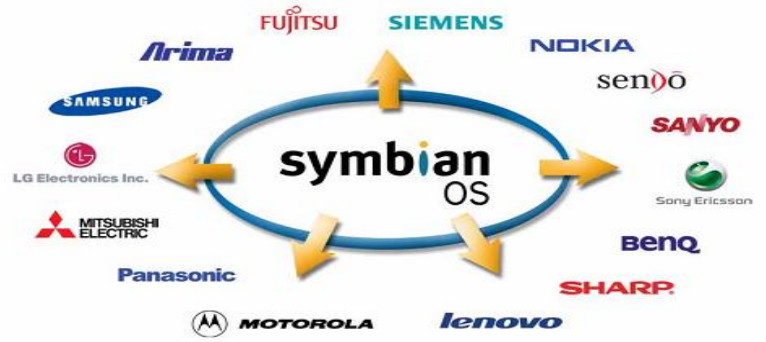
1.1 Giới thiệu hệ điều hành Symbian

Symbian là một công ty phần mềm chuyên phát triển và cung cấp một hệ điều hành tiên tiến, mở, chuẩn mực dùng cho thiết bị di động – hệ điều hành Symbian. Công ty được thành lập vào tháng 6 nam 1998 đặt trụ sở tại Anh. Mục tiêu của công ty Symbian là phát triển hệ điều hành Symbian thành hệ điều hành chuẩn được sử dụng rãi cho các hệ thống kĩ thuật số di động trên toàn thế giới. Đươc sự hậu thuẫn củaa các nhà sản xuất điện thoại di động hàng đầu thế giới, công ty Symbian không ngừng phát triển:



Các cổ đông của công ty Symbian

Ban đầu công ty Psion xây dựng EPOC platform dùng để điều khiển các thiết bị nhỏ, đạt được một số kết quả nhất định sau đó, các công ty điện thoại di đông hàng đầu (Nokia, Siemens…) mua lại Psion, thành lập công ty Symbian và tiếp tục phát triển EPOC với tên gọi hệ điều hành Symbian. Ngày nay, hệ điều hành Symbian là hệ điều hành được sử dụng rãi trên các thiết bị di động. Như cam kết cung cung cấp một chuẩn mở và hỗ trợ thiết bị người dùng di động mà Symbian trở thành sự lựa chọn hàng đầu trong ngành công nghiệp về thiết bị di động hiện nay. hệ điều hành Symbian là một chuẩn mở nên bất cứ nhà sản xuất nào cung có thể được cấp bản quyền sử dụng trên thiết bị của mình.



Các nhà sản xuất có giấy phép sử dụng hệ điều hành Symbian

1.2 Các mô hình thiết bị sử dụng hệ điều hành Symbian

Hệ điều hành Symbian được thiết kế cho hai loại thiết bị di động chiến lược là Communicator và Smartphone. Communicator là các máy PDA với khả năng liên lạc vô tuyến của thiết bị di động. Trong khi Smartphone là điện thoại di động với các tính nang PDA bổ sung. Với hai loại thiết bị này, Symbian công bố một số mô hình thiết kế tham khảo cho các nhà sản xuất. Hiện nay, tất cả các thiết bị di động thông minh trên thi trường đều có thể xác định dùng một trong ba mô hình sau:

Mô hình Crystal: Mô hình Crystal định nghĩa một loại Communicator bỏ túi với hình dáng của một máy laptop. Crystal sử dụng màn hình màu theo chuẩn ½ VGA và môt bàn phím QWERTY, có thể hỗ trợ màn hình cảm ứng để nhập liệu với bút stylus.

Mô hình Quartz: Mô hình Quartz định nghĩa một loại Communicator với hình dáng của một máy Pocket PC. Quartz sử dụng màn hình màu theo chu?n ¼ VGA, là một thiết bị di động dùng bút stylus nhập liệu qua tuong tác với một màn hình cảm ứng

Mô hình Pearl: Mô hình Pearl định nghia một loại Smartphone với hình dáng kích thước của một điện thoại di động thông thường. Pearl hỗ trợ màn hình màu với nhiều kích thước, tiêu chuẩn khác nhau, sử dụng bàn phím số của điện thoại để nhập liệu.

1.3 Các tiến trình và tiểu tiến trình trong Symbian

1.3.1 Tiến trình

Giống như các hệ điều hành khác, tiến trình (process) là đơn vị cơ sở cần bảo vệ trong symbian. Mỗi tiến trình có một khoảng không gian địa chỉ riêng. Địa chỉ ảo của chương trình khi thực thi sẽ đc ánh xạ thành các địa chỉ vật lí trên ROM (với các chương trình thực hiện trực tiếp trên ROM) và trên RAM (chứa các ma chương trình, dữ liệu động và các thành phần khác cần cho hoạt động của chương trình) tạo không gian bộ nhớ cho tiến trình. Công việc ánh xạ này được thực hiện bởi đơn vị quản lí bộ nhớ (Memory Management Unit – MMU). Do địa chỉ mã chương trình trên ROM luôn cố định nên các chương trình trên ROM có thể dùng chung (2 ứng dụng sử dụng 1 DLL lưu trên ROM). Còn trên RAM, mỗi tiến trình có một vùng nhớ riêng, không được truy xuất lẫn nhau.

1.3.2 Tiểu trình.

Tiểu trình (Thread) là đơn vị cơ sở thực thi chương trình trên symbian. Một tiến trình sẽ bao gồm nhiều tiểu trình và các tiểu trình này sử dụng cùng một vùng nhớ được cung cấp cho tiến trình đó. Thông thường trên symbian, một tiến trình có nhiều tiểu trình.

1.3.3 Tiến trình và tiểu trình nhân

Trong nhân, chỉ có duy nhất một tiến trình hoạt động: tiến trình nhân. Nó hoạt động ở chế độ dặc quyền. có 2 hoạt động phục vụ cho nhân: tiểu trình phục vụ nhân (Kernel sever) và tiểu trình rỗng (null). Tiểu trình phục vụ nhân là tiểu trình có mức độ ưu tiên cao nhất trong hệ thống. Bất cứ khi nào có yêu cầu sử dụng các dịch vụ hệ thống trong nhân là tiểu trình này lại hoạt động ngay lập tức. Nược lại, tiến trình null là tiểu trình có độ ưu tiên thấp nhất trong hệ thống. Tuy vậy nó lại đóng vai trò rất quan trọng trong hệ thống. khi điện thoại symbian khởi động, hệ điều hành bắt đầu hoạt động thì tiểu trình null là tiểu trình chạy đầu tiên với nhiệm vụ là nạp file sever. Khi hệ thống

đang hoạt động, tiểu trình null sẽ không bao giờ được gọi vì có độ ưu tiên thấp nhất. Nhưng khi hệ thống không còn phục vụ cho một ứng dụng nào nữa. tiểu trinh null sẽ được thực hiện. Nhiệm vụ của nó lúc này là gọi phần quản lí năng lượng để đưa hệ thống vào trạng thái "ngủ" để giảm thiểu hao hụt năng lượng

1.3.4 Quản lí và điều phối tiến trình

Việc điều phối và quản lý tiến trình, tiểu trình do nhân đảm trách. Bộ điều phối tiến trình hoạt động theo cơ chế độ ưu tiên với nguyên lý không độc quyền suer dụng thuật toán điều phối Round Rubin: trong một chu kì, tiểu trình có độ ưu tiên cao nhất sẽ được chạy trước tiên, các tiểu trình khác ở trạng thái tạm hoãn (suspend). Nhân hệ thống kiểm tra độ ưu tiên của các tiểu trình tại đầu chu kỳ và sẽ phục hồi hoạt động (resume) cho tiểu trình này nếu tiểu trình này có độ ưu tiên lớn hơn tiểu trình đang hoạt động.

Thông thường để xây dựng cơ chế quản lý sự kiện cho các tiến trình, các hệ điều hành sử dụng cơ chế đa tiểu trình (multi thread): ví dụ một tác vụ được tính toán lâu phức tạp được xử lí bởi một tiểu trình trong khi một tiểu trình khác tiếp tục được chờ xử lý các sự kiện nhập của người dùng. Cơ chế này symbian có hỗ trợ nhưng hiếm khi được dùng do bộ xử lí không mạnh mẽ như trên máy vi tính và sử dụng nhiều tiểu trình cũng không phù hợp với mô hình sử dụng sever trên symbian. Do đó symbian đã sử dụng một cơ chế tối ưu hơn cho hoạt động của ứng dụng và các sever: đó là mọi vấn đề quản lý sự kiện và xủ lý tác vụ đồng thời đều được thực hiện nhờ một đối tượng đặc biệt trong symbian, active object. Mỗi một tiểu trình trên hệ điều hành symbian có một bộ điều phối active scheduler đảm trách việc quản lý sự kiện thông qua việc quản lý điều phối một hoặc nhiều active object.

CHƯƠNG 2: Kĩ thuật lập trình C++ trên Symbian

2.1 Các kiểu dữ liệu cơ bản

Môi trường lập trình trên Symbian cung cấp các kiểu dữ liệu cơ bản tương đương với các kiểu xây dựng sẵn của C++. Đó là các lớp dữ liệu cơ bản mà khi sử dụng không cần cấp phát hay hủy 1 cách tường minh; các lớp này bắt đầu bằng kí tự T. Lưu ý, khi lập trình trên Symbian không nên dùng các kiểu dữ liệu xây dựng sẵn của C++ mà hãy dùng các (lớp) kiểu co bản mà Symbian cung cấp. Lý do đơn giản vì thiết bị thực (chạy hệ điều hành Symbian) có thể không làm việc tốt với các kiểu dữ liệu xây dựng sẵn của C++.

kiểu số nguyên:

Kiểu dữ liệu có dấu	Kiểu dữ liệu không dấu	Kiểu dữ liệu C++ tương đương	Mô tả
TInt8	TUint8	signed char & unsigned char	Số nguyên 8 bit có dấu và không dấu
TInt16	TUint16	short int & unsigned short int	Số nguyên 16 bit có dấu và không dấu
TInt32	TInt32	long int & unsigned long int	Số nguyên 32 bit có dấu và không dấu
TInt64			Số nguyên 64 bit
TInt	TInt	int & unsigned int	Số nguyên ít nhất 32 bit có dấu và không dấu

Kiểu số thực:

Kiểu dữ liệu	Kiểu dữ liệu C++ tương đương	Mô tả
TReal32	float	Số thực 32 bít
TReal64	double	Số thực 64 bít
TReal	Tương đương với TReal64	

Lưu ý: Hầu hết các HĐH Symbian hiện nay không hỗ trợ phần cứng xử lý số chấm động. Vì vậy các phép toán trên số thực sẽ chậm hon rất nhiều lần so với số nguyên. Vì vậy bạn nên hạn chế tôi đa việc sử dụng số thực.

Các kiểu cơ bản khác:

Kiểu dữ liệu	Kiểu dữ liệu C++ tương đương	Mô tả
TChar	32-bit unsigned integer	Kiểu ký tự của Symbian, dài 32 bit, cung cấp nhiều hàm xử lý trên ký tự
TText8	char	Kiểu ký tự 1 byte
TText16	wchar_t	Kiểu ký tự Unicode (2 bytes)
Ttext	wchar_t	Kiểu ký tự Unicode (2 bytes)
Tbool	int	Kiểu logic, nhận 2 giá trị ETrue/ EFalse
TAny	void	

2.2 Quản lí lỗi

2.2.1 Cơ chế bắt lỗi cơ bản mà Symbian hỗ trợ gồm:

- Hàm User::Leave() có tác dụng ngừng hàm đang thực hiện và trả về mã lỗi.

- Macro TRAP và biến thể của nó TRAPD, cho phép đoạn mã chương trình hoạt động dưới dạng bẫy lỗi

Cơ chế này hoạt động như thao tác bẫy lỗi khá quen thuộc try/catch và thrown

Tất cả các hàm có thể phát sinh lỗi tài nguyên trong ứng dụng (nhu cấp phát vùng nhớ cho một đối tượng nhưng đã hết bộ nhớ, truyền dữ liệu trong khi dịch vụ chưa sẵn sàng…) gọi là "Leave function", và có ký tự L cuối tên hàm (hàm L). Lưu ý một hàm có thể ngừng (leave) 1 cách trực tiếp do đoạn mã phát sinh lỗi (leave directly) hoặc do 1 hàm này gọi 1 hàm L khác và hàm L được gọi này có thể gây ra lỗi (leave indirectly).

Ta có thể sử dụng tiện ích User::Leave() đê ngừng ngay hàm đang thực hiện và trả về mã lỗi tương ứng (một dạng biệt lệ (exception) trong Symbian). Ví dụ dưới đây cho thấy hàm sẽ ngừng và trả về mã lỗ thiếu bộ nhớ nếu việc cấp phát thông qua toán tử new không thành công:

```
void doExampleL()
      {
     CExample* myExample = new CExample;
      if (!myExample) User::Leave(KErrNoMemory);
     // leave used in place of return to indicate an error
     // if leave, below code isn't executed
     // do something
     myExample->iInt = 5;
     testConsole.Printf(_LIT("Value of iInt is %d.\n"), myExample-
>iInt);
     // delete
     delete myExample;
     }
```

Tất cả những hàm leave, gồm cả những hàm gọi hàm leave khác đều hoạt động dưới cơ chế bẫy lỗi. Nếu hàm User::Leave() được gọi thì dòng điều khiển chương trình được trả về cho macro TRAP gần nhất. Một biến được dùng dể nhận giá trị lỗi trả về của User::Leave() và nếu không có lỗi gì xảy ra thì biến này sẽ có giá trị KErrNone. (Lưu ý biến trả về cảa macro TRAP không phải là giá tri hàm trả về).

Xét ví dụ cụ thể sau:

```
TInt E32Main()
      {
         TInt r; // The leave variable
      // Perform example function. If it leaves,
      // the leave code is put in r
      TRAP(r,doExampleL());
      if (r)      // Test the leave variable
      testConsole.Printf(_LIT("Failed: leave code=%d"), r);
      }
```

Macro TRAP có 1 biến thể khác cho phép rút gọn code chương trình đó là TRAPD, khi sử dụng TRAPD ta không cần khai báo biến lỗi một cách tường minh:

TRAPD(leaveCode,value=GetSomethingL()); // get a value

Ngoài ra Symbian còn cung cấp 1 dạng toán tử new mới, sủ dụng cơ chế leave trong 1 dòng lệnh, dó là: new (ELeave)… Khi sử dụng new (ELeave)… nghĩa là nếu việc cấp phát vùng nhớ cho 1 đối tượng không thành công thì hàm thực thi toán tử new này sẽ ngừng ngay lập tức:

```
void doExampleL()
{
 // attempt to allocate, leave if could not
 CExample* myExample = new (ELeave) CExample;
// new (ELeave) replaces new followed by check
 // do something
 myExample->iInt = 5;
 // delete
 delete myExample;
}
```

Trong ví dụ trên, nếu việc cấp phát myExample không thành công hàm doExampleL() sẽ ngừng ngay và trả về giá trị lỗi cho macro TRAP hay TRAPD gọi hàm này (nếu có).

2.2.2 Hàm Cleanup stack

Khi một hàm leave, dòng điều khiển được chuyển cho macro TRAP và dòng lệnh dưới TRAP được thực thi. Điều này nghĩa là những đối tượng được tạo ra hoặc truyền vào trước khi hàm ngừng sẽ trở nên "mồ côi', việc hủy chúng không được thực hiện và tài nguyên (bộ nhớ) mà chúng chiếm giữ sẽ

không được giải phóng. Hệ điều hành Symbian cung cấp một cơ chế quản lý những đối tượng này gọi là Cleanup stack. Như đã trình bày ở phần qui ước đặt tên, chỉ có các lớp bắt đầu bằng ký tự C là cần và bắt buộc phải hủy khi sử dụng xong.

Như vậy nguy cơ đối tượng mồ côi xuất phát từ lớp C, xét ví dụ cụ thể sau:

```
void doExampleL()
 {
 // An T-type object: can be declared on the stack
 TBuf<10> buf;
 // A C-type object: must be allocated on the heap
 // Allocate and leave if can not
 CExample* myExample = new (ELeave) CExample;
 // do something that cannot leave: no protection needed
 myExample->iInt = 5;
 // PROBLEM: do something that can leave !!!
 myExample->DoSomethingL();
 // delete
 delete myExample;
 }
```

Nếu lúc này hàm myExample->DoSomethingL(); bị lỗi và ngừng lại thì dòng lệnh delete myExample sẽ không được thực thi, nghia là vùng nhớ cấp cho nó không được giải phóng. Ta giải quyết vấn đề này bằng ki thuật Cleanup stack mà Symbian hỗ trợ nhu sau: dùng hàm CleanupStack::PushL() để dua con trỏ đối tượng vào Cleanup stack trước khi gọi bất kì hàm leave nào; sau đó nếu những hàm leave đã hoàn thành mà không có lỗi xảy ra, ta gọi hàm CleanupStack::Pop() để lấy con trỏ đối tượng ra khỏi Cleanup stack.

Ví dụ minh họa:

```
void doExampleL()
{
// allocate with checking
CExample* myExample = new (ELeave) CExample;
// do something that cannot leave
myExample->iInt = 5; // cannot leave: no protection needed
// do something that can leave: use cleanup stack
CleanupStack::PushL(myExample); // pointer on cleanup stack
myExample->DoSomethingL(); // something that might leave
CleanupStack::Pop(); // it didn't leave: pop the pointer
// delete
delete myExample;
}
```

Lưu ý: hàm CleanupStack::PushL() có thê leave, tuy nhiên nếu hàm này leave thì đối tượng đưa vào stack bởi hàm này cung sẽ bị hủy.

Cần luu ý hàm CleanupStack::Pop() tự nó không có ý nghĩa gì, nó chỉ đẩy 1 đối tượng ra khỏi Cleanupstack và ta có thể hủy hoặc sử dụng đối tượng nhận được từ stack này sau khi hàm bị ngừng (leave) đảm bảo không có đối tượng "mồ côi".

Trong nhiều trường hợp ta có thể gọi hàm CleanupStack::PopAndDestroy() để hủy đối tượng sau khi hoàn thành hàm leave:

```
void doExampleL()
{
. . .
// pop from cleanup stack, and destroy, in one operation
CleanupStack::PopAndDestroy(); //don't need call delete
}
```

Để rút gọn mã nguồn chương trình, việc cấp phát và đưa đối tượng vào Cleanp stack thường được thực hiện trong 1 hàm kết thúc bằng ký tự C. Ví dụ hàm TAny* User::AllocLC() gồm việc gọi hàm User::Alloc() để cấp phát vùng nhớ sau đó leave nếu cấp phát không thành công, ngược lại tự động đưa đối tượng vào Cleanup stack. Nếu gọi hàm ___C để cấp phát đối tượng vẫn phải lấy đối tượng ra khỏi Cleanup stack sau khi sử dụng xong.

2.2.3 Hàm dựng 2 pha

Trong khi khởi tạo các đối tượng phức tạp (đối tượng lớp C chứa một hoặc nhiều đối tượng lớp C khác cần được khởi tạo) có thể dẫn tới tình trạng leave mà việc quản lý bằng Cleanup stack một cách tường minh gặp khó khăn do ta không biết đối tượng nào đã được khởi tạo và đối tượng nào chưa. Lúc này Symbian tiếp tục cung cấp cơ chế quản lý lỗi khi khởi tạo đối tượng gọi là hàm dựng 2 pha (2 phase constructor) hoạt động với cơ chế như sau:

- Cấp phát vùng nhớ cho đối tượng (và leave nếu không đủ bộ nhớ)
- Khởi tạo các thành phần an toàn (không thể leave)

- Đưa con trỏ đối tượng vào Cleanup stack

- Dùng hàm dựng thứ 2 (2nd phase constructor) để khởi tạo các thành phần có thể leave.

Luu ý:

Toàn bộ quá trình trên được thực hiện thông qua 2 hàm tinh: NewL(), and NewLC()(tự đưa đối tượng được cấp phát vào Cleanup stack)

Hàm dưng thứ 2 có tên là ConstructL().

Ví dụ ta có lớp CSimple là một lớp đơn giản không chứa các đối tượng khác:

```
class CSimple : public CBase
{
public:
      CSimple(TInt); //hàm d?ng
      void Display();
private:
      TInt iVal;
};
```

Lớp CCompound chứa đối tượng CSimple như là biến thành viên

```
class CCompound : public CBase
{
public:
      void Display();
      ~CCompound();
      static CCompound* NewL(TInt aVal);
      static CCompound* NewLC(TInt aVal);
protected:
      CCompound(TInt aVal);
      void ConstructL();
private:
      TInt iVal;
      CSimple* iChild;
 };
```

Minh họa nguy cơ từ việc khởi tạo đối tượng CCompound theo cách thông thường:

```
CCompound::CCompound(TInt aVal)
      {
      iVal=aVal;
      iChild = new (ELeave) CSimple(aVal);
      }
```

Lúc này, khi khởi tạo 1 đối tượng CCompound , nếu vì lý do nào đó mà việc cấp phát đối tượng iChild (CSimple) không thành công. Hàm khởi tạo CCompound(TInt aVal) bị ngừng (leave). Lúc này đối tượng CCompound đã được cấp phát, tuy nhiên không thể truy cập đến vùng nhớ này vì hàm dựng không thành công. Như vậy đã có một đối tượng mồ côi được tạo ra trong vùng nhớ mà ta không thể quản lý được.

Để khắc phục CCompound sử dụng hàm dưng 2 pha, có thể tạo ra một đối tượng CCompound một cách an toàn thông qua hai hàm tính NewL() và NewLC() như sau:

```
// NewLC with two stage construction
CCompound* CCompound::NewLC(TInt aVal)
     {
      // get new, leave if can't
      CCompound* self=new (ELeave) CCompound(aVal);
      // push onto cleanup stack in case self->ConstructL leaves

      // complete construction with second phase constructor
      self->ConstructL();
      return self;
      }
void CCompound::ConstructL()
      {
      // function may leave, as creating a new CSimple object
      // may leave.
      iChild = new (ELeave) CSimple (iVal);
      }
CCompound* CCompound::NewL(TInt aVal)
      {
      CCompound* self=NewLC(aVal);
      CleanupStack::Pop();
      return self;
      }
```

Ta sử dụng hàm NewL(), NewLC() để khởi tạo đối tượng CCompound an toàn như sau:

```
void doExampleL()
      {
       // allocate and push to cleanup stack - leave if failed
       CCompound* myExample = CExample::NewLC(5);
        // do something that might leave
        myExample->DoSomethingL();
       // pop from cleanup stack and destroy
       CleanupStack::PopAndDestroy();
      }
```

Lúc này nếu hàm doExampleL() không thành công do không cấp phát được đối tượng CSimple (biết được bằng bẫy lỗi TRAPD(errcode, doExampleL()) với errcode != KErrNone), ta chỉ cần gọi CleanupStack::PopAndDestroy() để "làm sạch" vùng nhớ.

Ngoài ra, đối với các lớp T, R Cleanup stack cung hỗ trợ thao tác push và pop thông qua các hàm overload:

```
static void PushL(TAny* aPtr);
static void PushL(TCleanupItem anItem);
```

CHƯƠNG 3: OPENGL ES

3.1 Giới thiệu về OpenGL ES

OpenGL ES là một sản phẩm miễn phí bao gồm các hàm API cho phép tạo các ứng dụng 2D, 3D trên các ứng dụng nhúng – bao gồm các thiết bị cầm tay. Nó được định nghĩa như là một tập con của openGL, tạo ra tính linh hoạt, mạnh mẽ trên giao diện cấp thấp giữa các phần mềm và đồ họa. OpenGL ES 1.1 nhấn mạnh về tốc độ phần cứng của các hàm API, trong khi OpenGL ES 1.0 chỉ tập trung vào các phần mềm cho phép triển khai. OpenGL ES 1.1 hoàn toàn tương thích với bản OpenGL ES 1.0 và nó có thể dễ dang thêm các API giữa hai phiên bản

Các đặc điểm của OpenGL ES được phát triển bởi nhóm Khronos

3.2 Nhập dữ liệu từ phím (Keyboard Input)

Đầu tiên bạn phải xây dựng một chức năng để xử lí mọi dữ liệu được đưa vào từ bàn phím,chức năng này phải chấp nhận một số các tham số nhất định

- Tham số thứ nhất là biến UGWindow
- Tham số thứ hai phải là một biến nguyên (interger), đại diện cho phím đã được bấm
- Tham số thứ ba và thứ tư cũng là hai biến nguyên (interger), xác định giá trị x, y của con trỏ thiết bị khi được ấn.

```
void keyboard(UGWindow uwin, int key, int x, int y)
{
// kiểm tra nút đã được bấm
 switch(key)
      {
      case 'q' : exit(0); break;
              // Các  phím có sẵn được liệt kê ở bảng dưới
```

Identifier	Description
UG_KEY_F1 - UG_KEY_F2	F1 through F12 keys.
UG_KEY_LEFT	Left arrow
UG_KEY_RIGHT	Right arrow
UG_KEY_UP	Up arrow
UG_KEY_DOWN	Down arrow
UG_KEY_PAGE_UP	Page Up key
UG_KEY_PAGE_DOWN	Page Down key
UG_KEY_HOME	Home key
UG_KEY_END	End key
UG_KEY_INSERT	Insert key

```
// thoát khỏi chương trình nếu ấn phím mũi tên đi lên
      case UG_KEY_UP : exit(0); break;
      }
}
```

3.3 Dựng (Rendering)

Các bước khởi tạo và thiết lập OpenGL ES, khi vẽ trên màn hình OpenGL ES sử dụng kĩ thuật của một bộ đêm kép. Khi vẽ chúng ta vẽ trên bộ nhớ đệm. Sau khi có được tất cả các thông tin của việc vẽ, nó sẽ trao đổi giữa các bộ nhớ đệm và bắt đầu vẽ trên bộ nhớ đệm khác. Điều này để ngăn chặn ảnh hưởng của việc chớp màn hình bởi hằng số xóa màn hình và vẽ hình khác trên một bộ nhớ đệm.

Trong hàm Init(), chúng ta sử dụng một lời gọi glClearColor, nó được sử dụng để xác định màu sắc cho màn hình hiện thị, nó bao gồm 4 tham số, các tham số này đại diện cho hệ màu RGBA và có giá trị trong khoảng từ 0 đến 1. Ba tham số đầu là màu đỏ xanh lá cây và xanh da trời, còn tham số thứ 4 là độ sáng tối của window

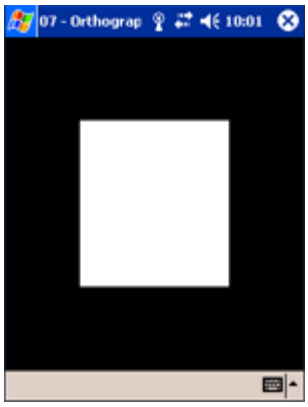
Đoạn code đặt màu nền đen cho màn hình hiển thị

Nội dung của hàm main.cpp

```
void init()
{
      glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}
```

```
void display(UGWindow uwin)
{
      // Bây giờ hãy thay đổi các giá trị của màu xem thử!Hàm glClear()
mới thực sự xoá window, nó có những hằng số xác định
      glClear(GL_COLOR_BUFFER_BIT);
      // Có trường hợp có những hàm chưa được chạy đến khi kết thúc
chương trình, để tránh trường hợp này hàm glFlush()được gọi, nó sẽ thực
hiện tất cả các hàm chưa được chạy và kết thúc chương trình.
         glFlush();
      //lưu thông tin sau khi vẽ trên khung, chúng trao đổi giữa các bộ
nhớ đệm và bắt đầu vẽ trên đó. Chức năng ugSwapBuffers được sử dụng để
thực hiện điều này
        ugSwapBuffers(uwin);

}
```

3.4 Phép chiếu trực giao (Orthographic Projection)



Có hai cách để hiển thị đối tượng đó là sử dụng phép chiếu phối cảnh và phép chiếu trực giao

Phép chiếu trực giao, view volume được định nghĩa là một hình hộp chữ nhật, vật thể nằm trong view volume được chiếu trực giao lên khung nhìn do đó trong phép chiếu trực giao khoảng cách từ camare đến vật thể không ảnh hưởng đến độ lớn của ảnh.

Trong phần này chúng ta sẽ tìm hiểu làm thế nào để hiển thị một hình lên màn hình, hình được tạo ra bằng cách xác định các đỉnh, đây là những điểm trong không gian 3 chiều vì vậy cần chỉ rõ các điểm trên hình.

Danh sách các tham số

Primitive Flag	Description
GL_POINTS	Các điểm
GL_LINES	Đoạn thẳng
GL_LINE_STRIP	Đường gấp khúc không khép kín
GL_LINE_LOOP	Đường gấp khúc khép kín
GL_TRIANGLES	Tam giác
GL_TRIANGLE_STRIP	Một dải tam giác được liên kết với nhau
GL_TRIANGLE_FAN	Các tam giác liên kết theo hình quạt

Khi vẽ điểm, chức năng **glPointSize** có thể thay đổi kích cỡ của điểm được vẽ, kích cỡ mặc định là 1.

Khi vẽ đường bạn có thể sử dụng **glLineWidth** để xác định độ rộng của đường, kích cỡ mặc định là 1.

Nội dung của hàm main.cpp

Bước đầu tiên là xác định tọa độ của hình vuông đặt trên màn hình, thiết lập 3 giá trị (float) x, y, z cho mỗi đỉnh

```
GLfloat square[] = {
      0.25, 0.25, 0.0,
      0.75, 0.25, 0.0,
      0.25, 0.75, 0.0,
      0.75, 0.75, 0.0
   };
```

Khởi tạo chương trình

```
void init()
{
      glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

Thiết lập chế độ ma trân bằng câu lệnh **glMatrixMode**(GL_PROJECTION) trước khi định nghĩa phép chiếu

```
      glMatrixMode(GL_PROJECTION);
```

Thiết lập ma trận hiện thời về ma trận đơn vị bằng lệnh **glLoadIdentity**()

```
      glLoadIdentity();
```

Ở phần đầu của hướng dẫn, chúng ta sử dụng phép chiếu trực giao. Chức năng **glOrthof** được chỉ định để xác định nhìn theo phép chiếu trực giao, nó bao gồm **glOrthof**(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top,GLfloat near, GLfloat far)

```
      glOrthof(0.0f, 1.0f, 0.0f, 1.0f, -1.0f, 1.0f);
```

Đến giờ ta đã thiết lập xong phép chiếu trực giao, tiếp đến ta sẽ vẽ hình bằng cách sử dụng chức năng glVertexPointer, chức năng này có 4 tham số:

- **GLint** size**: Xác định số lượng tọa độ cho mỗi đỉnh**
- **GLenum** type: Xác định kiểu dữ liệu của mỗi đỉnh trong mảng vi dụ như GL_BYTE, GL_SHORT, GL_FLOAT v.v…

- **GLsizei** stride: Xác định khoảng cách byte giữa các đỉnh liên tiếp, Nếu
- stride bằng 0 các đỉnh được hiểu là đã được đóng gói chặt chẽ trong mảng , giá trị ban đầu bằng 0
- **const GLvoid ***pointer**:** Xác định vị trí bộ nhớ của giá trị đầu tiên trong mảng, nó trỏ tới mảng.

```
    glVertexPointer(3, GL_FLOAT, 0, square);
```

Chức năng **glEnableClientState** sẽ đua ra một trong những tham số chỉ định mảng đó phải được kích hoạt

```
    glEnableClientState(GL_VERTEX_ARRAY);
}
```

Bây giờ chúng ta có thể thiết lập chế độ hiển thị, hãy nhớ rằng bạn đang sử dụng thư viện Vincent, màn hình hiển thị chức năng cần phải chấp nhận một tham số UGWindow

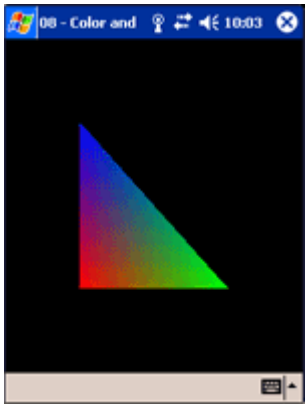
```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
```

Chức năng glDrawArray với cac tham số

- **GLenum** mode: xác định giá trị ban đầu để vẽ
- **GLint** first: Xác định chỉ số ban đầu của mảng
- **GLsizei** count: chỉ rõ số đỉnh để xử lý

```
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
  glFlush();
  glutSwapBuffers();
}
```

3.5 Màu sắc và đánh bóng (Color and Shading)



Tất cả màu sắc trong OpenGL được đại diện bởi 4 giá trị, 3 giá trị màu đỏ, xanh lá cây và xanh lam, cuối cùng là giá tri alpha, điều này chỉ thể hiện rõ ràng 1 màu. Điều này sẽ được nói rõ hơn trong phần này.

Ta sẽ sử dụng một mảng màu.

Nội dung của hàm main.cpp

Ta sẽ khởi tạo một mảng tam giác.

```
GLfloat triangle[] = {
        0.25f, 0.25f, 0.0f,
        0.75f, 0.25f, 0.0f,
        0.25f, 0.75f, 0.0f
};
```

Tiếp theo ta sẽ tạo ra một mảng màu. Chúng tôi cung cấp cho mỗi đỉnh một màu sắc khác nhau, màu đỏ, xanh lá cây va xanh lam

```
GLfloat colors[] = {
        1.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f
};
```

Một biến boolean shaded được tạo để theo dõi xem có được đánh bóng hay không, chúng tôi sử dụng biến này để chuyển đổi giữa việc tô bóng hay không tô bóng hình

```
bool shaded = false;
```

Thiết lập phép chiếu trực giao

```
void init()
{
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrthof(0.0f, 1.0f, 0.0f, 1.0f, -1.0f, 1.0f);
        glVertexPointer(3, GL_FLOAT, 0, triangle);
```

Ta sử dụng hàm **glColorPointer** đê thiết lập cho mảng màu, hàm này làm việc giống như chức năng **glVertexPointer**, chúng có 4 tham số và tham số đầu tiên để xác định có 4 float (một giá trị màu) cho mỗi đỉnh .

```
        glColorPointer(4, GL_FLOAT, 0, colors);
```

Chúng ta phải kích hoạt các đỉnh và mảng màu

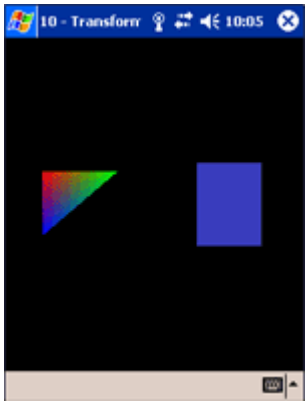
```
        glEnableClientState(GL_VERTEX_ARRAY);
        glEnableClientState(GL_COLOR_ARRAY);
```

Bây giờ ta thêm màu và shading (tô bóng) vào hình. Có 2 loại shading. Điều này được xác định bằng cách sử dụng chức năng **glShadeModel**, chức năng này sẽ đưa ra một trong hai tham số GL_FLAT và GL_SMOOTH để xác định loại shading và GL_SMOOTH được thiết lập theo mặc định.

```
    glShadeModel(GL_FLAT);
}
```

Thiết lập chế độ màn hình hiển thị (như phần trước) chỉ khác trong lời gọi chức năng **glDrawArrays** ta sử dụng cờ GL_TRIANGLES để vẽ 3 đỉnh của tam giác.

3.6 Phép biến đổi (Transformations)



Phần này sẽ giới thiệu về cách chuyển đổi hình theo các cách khác nhau

1. Phép tỉ lệ - **glScalef**
2. Phép dịch - **glTranslatef**
3. Phép quay - **glRotatef**

Nội dung của hàm main.cpp

Khởi tạo 2 biến dùng để quay theo trục x và y

```
float xrot = 0.0f;
float yrot = 0.0f;
```

Ta sẽ khởi tạo một tam giác.

```
GLfloat triangle[] = {
      -0.25f, -0.25f, 0.0f,
      0.25f, -0.25f, 0.0f,
      -0.25f, 0.25f, 0.0f
};
```

Khởi tạo một mảng màu

```
 GLfloat square[] = {
       -0.25f, -0.25f,
       0.25f, -0.25f,
       -0.25f, 0.25f,
       0.25f, 0.25f
 };
```

Chức năng Init chỉ để gọi hàm **glClearColor**

```
void init()
{
      glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}
```

Thiết lập chế độ hiển thi

```
void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
```

Ta sẽ vè một tam giác ở phía bên trái màn hình và một hình vuông ở phía bên phải, tam giác sẽ được tô bóng mịn và hình vuông sẽ được tô bóng.

Các thiết lập trên tam giác cũng như bài trước

```
// Triangle
glShadeModel(GL_SMOOTH);

glVertexPointer(3, GL_FLOAT, 0, triangle);
glColorPointer(4, GL_FLOAT, 0, colors);

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
```

Ta sẽ thay đổi đoạn code như sau. Nhớ rằng trong hàm reshape chúng ta đặt đối tưởng hiển thi như ma trận hiện thời. Ma trận được sử dụng cho các phép biến đổi. Có 3 phép chuyển đổi sử dụng bởi các hàm **glTranslatef, glScalef** và **glRotatef.** Các giá trị f ở cuối mỗi hàm thể hiện biến đầu vào mang giá trị float.

Sau khi vẽ tam giác chúng ta không muốn các hình sau đó bị ảnh hưởng bởi việc chuyển đổi. Chức năng **glPushMatrix** và **glPopMatrix** được sử dụng để sao chép thêm một ma trận hiện thời đưa lên đỉnh ngăn xếp và loại bỏ ma trận hiện thời ra khỏi ngăn xếp.

Ví dụ: ta muốn vẽ 1 chiếc ô tô co 4 bánh, quá trình vẽ được mô tả như sau: vẽ thân xe, ghi nhớ bạn ở đâu, tịnh tiến về bánh xe phải phía trước, vẽ bánh xe, quay lại vi trí bạn đã ở (đưa thân xe về vị trí trước khi tinh tiến) ghi nhớ bạn đã ở đâu, tịnh tiến bánh xe trái phía trước….

```
glPushMatrix();
```

Hàm **glTranslatef** với 3 tham số cho truc x, y, z để dịch chuyển đối tượng (dịch sang trái 0.25 đơn vị và lên 0.5 đơn vi)

```
glTranslatef(0.25f, 0.5f, 0.0f);
```

Hàm **glScalef** với 3 tham số xác định tỉ lệ của đối tượng theo 3 trục x, y, z (giảm kích thước của tam giác xuống một nửa)

```
glScalef(0.5f, 0.5f, 0.5f);
```

Hàm **glRotatef** với 4 tham số là góc quay và 3 tham số đại diện cho 3 trục x, y, z để quay đối tượng (quay đối tượng theo 1 góc xrot theo trục x)

```
glRotatef(xrot, 1.0f, 0.0f, 0.0f);
```

Vẽ tam giác

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Phục hồi ma trân về thời điểm ban đầu

```
glPopMatrix();
```

Tiếp theo chúng tôi sẽ không sử dụng mảng màu cho hình nên sẽ khóa chức năng này lại

```
glDisableClientState(GL_COLOR_ARRAY);
```

Tiếp theo ta sẽ vẽ một hình vuông được tô bóng

```
// Square
glShadeModel(GL_FLAT);
```

Chú ý rằng khi chúng tôi khởi tạo con trỏ đỉnh, chúng tôi sử dụng 2 tham số đầu tiên đại diện cho mỗi đỉnh.

```
glVertexPointer(2, GL_FLOAT, 0, square);
```

Thay vì sử dụng mảng màu ta có thể sử dụng chức năng **glColor4f** hoặc **glColor4x**

```
glColor4f(0.25f, 0.25f, 0.75f, 1.0f);
```

Việc chuyển đổi hình vuông cũng tương tự như hình tam giác phía trên

```
    glPushMatrix();
        glTranslatef(0.75f, 0.5f ,0.0f);
        glScalef(0.5f, 0.5f, 0.5f);
        glRotatef(yrot, 0.0f, 1.0f, 0.0f);
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glPopMatrix();
    glFlush();
    glutSwapBuffers();
}
```

Để cho phép tạo ra hình ảnh động chúng tôi sử dụng chức năng **idle**, chức năng này được gọi là vòng lặp chính trong khi không có thông điệp nào đang được xử lý.

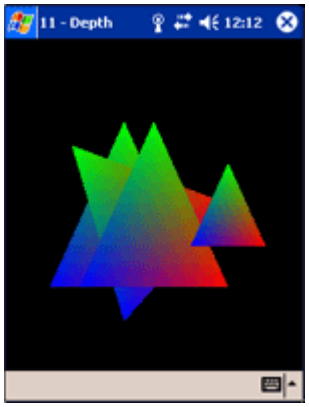
Chúng tôi muốn tăng góc quay của đối tượng trên trục x và truc y cũng như vẽ lại màn hình sau khi thay đổi. điều này được thực hiện khi gọi hàm **glutPostRedisplay** hoặc **ugPostRedisplay**

```
void idle()
{
	xrot += 2.0f;
	yrot += 3.0f;

	glutPostRedisplay();
}
```

Bước cuối cùng là ta sẽ thông báo cho thư viện GLUT|ES / UG là chức năng idle được sử dụng. Điều này được hoàn thành với lời gọi hàm **glutIdleFunc / ugIdleFunc**

```
	glutIdleFunc(idle);
```

3.7 Chiều sâu (Depth)



Trong phần này chúng ta sẽ thảo luận làm thế nào để thêm chiều sâu vào chương trình của bạn cho phép các z (trục) có thể phối hợp hoạt động một cách chính xác

Điều này được hoàn thành khi sử dụng lời gọi đến depth buffer. depth buffer có chứa một giá trị cho mỗi điểm ảnh trên màn hình, giá trị này trong khoảng từ 0 đến 1. Điều này đại diện cho khoảng cách từ đối tượng đến người xem, mỗi sự đồng bộ có sự liên kết sâu về giá trị. Khi hai giá trị chiều sâu được so sánh thì giá trị thấp hơn sẽ được hiển thị trên màn hình.

Nội dung của hàm main.cpp

```
void init()
{
```

Bước đầu tiên ta phải bật chức năng depth buffer điều này được thực hiện thông qua cờ GL_DEPTH_TEST trong hàm **glEnable**

```
	glEnable(GL_DEPTH_TEST);
```

Như phần đầu của hướng dẫn chúng tôi đã nói đến việc hạ thấp hơn giá trị của chiều sâu, sự phối hợp chặt chẽ hơn cho người xem. Điều này có thể thay đổi

bằng cách sử dụng chức năng **glDepthFunc** chức năng này chỉ định giá trị trong depth buffer để so sánh. Các giá trị này được thông báo qua bảng sau:

Cờ	Mô tả
GL_NEVER	Không bao giờ đi qua
GL_LESS	Đi qua nếu giá trị chiều sâu đưa vào nhỏ hơn giá trị được lưu trữ
GL_EQUAL	Đi qua nếu giá trị chiều sâu đưa vào bằng giá trị được lưu trữ
GL_LEQUAL	Đi qua nếu giá trị chiều sâu đưa vào nhỏ hơn hoặc bằng giá trị được lưu trữ
GL_GREATER	Đi qua nếu giá trị chiều sâu đưa vào lớn giá trị được lưu trữ
GL_NOTEQUAL	Đi qua nếu giá trị chiều sâu đưa vào không bằng giá trị được lưu trữ
GL_GEQUAL	Đi qua nếu giá trị chiều sâu đưa vào lớn hơn hoặc bằng giá trị được lưu trữ
GL_ALWAYS	Luôn đi qua

Giá trị cờ mặc định là GL_LESS chúng tôi muốn thử đi qua khi các giá trị bằng nhau. Điều này sẽ sảy ra khi các đối tượng có cùng các giá trị z, màn hình sẽ hiển thị tùy thuộc vào thứ tự mà đối tượng đó được in ra.

```
glDepthFunc(GL_LEQUAL);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

Sự thử chiều sâu để so sánh các giá trị bạn phải khởi tạo tất cả các giá trị trong bộ đệm. Điều này có thể đạt được bằng cách sử dụng chức năng **glClearDepthf**, chức năng này sẽ đưa ra một trong những tham số chỉ ra giá trị về chiều sâu trong bộ đệm dùng để khởi tạo cùng.

```
glClearDepthf(1.0f);
```

Hiển thị một số hình tam giác trên màn hình làm việc với depth buffer

```
        glVertexPointer(3, GL_FLOAT, 0, triangle);
        glColorPointer(4, GL_FLOAT, 0, colors);
        glEnableClientState(GL_VERTEX_ARRAY);
        glEnableClientState(GL_COLOR_ARRAY);
}

void display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glDrawArrays(GL_TRIANGLES, 0, 3);
```

Vẽ tam giác thứ 2 hơi ở trên tam giác đầu tiên

```
        glPushMatrix();
        {
                glTranslatef(-0.2f, 0.0f, -1.0f);

                glDrawArrays(GL_TRIANGLES, 0, 3);
```

Tam giác thứ 3 quay 45 độ theo trục z của tam giác thứ 2

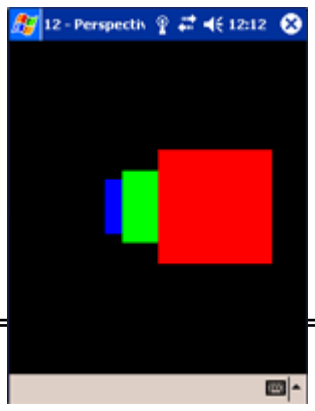
```
                glRotatef(45.0f, 0.0f, 0.0f, 1.0f);

                glDrawArrays(GL_TRIANGLES, 0, 3);
        }
        glPopMatrix();
```

Cuối cùng là tam giác đặt cùng với trục z của tam giác đầu tiên, đây là hình tam giác nhỏ nằm ở phía bên phải.

```
        glPushMatrix();
        {
                glTranslatef(0.5f, 0.0f, 0.0f);
                glScalef(0.5f, 0.5f, 0.5f);

                glDrawArrays(GL_TRIANGLES, 0, 3);
        }
        glPopMatrix();

        glFlush();
        glutSwapBuffers();
}
```

3.8 Hình phối cảnh (Perspective)

Trong thế giới thực, nếu bạn có nhiều đối tượng có cùng một kích cỡ được đặt ở những khoảng cách khác nhau, bạn sẽ nhận thấy rằng các đối tượng ở xa hơn thì sẽ trông nhỏ hơn.



Trong phần hướng dẫn trước bạn có thể nhận thấy rằng các tam giác phía sau thực sự có cùng kích thước với tam giác đầu tiên khi nhìn.

Trong phần hướng dẫn này sẽ giải thích cách làm cho các đối tượng ở xa hơn thì sẽ trông nhỏ hơn, chúng ta cũng sẽ thảo luận hình dạng thế nào là đạt tiêu chuẩn bằng cách sử dụng thư viên UG.

Nội dung của hàm main.cpp

Đầu tiên chúng tôi sẽ tạo 2 biến để giữ cho chiều rộng và chiều cao của cửa sổ, bạn sẽ thấy nó được sử dụng thế nào sau này.

```
int w = 0;
int h = 0;
```

Một biến dể giữ để xác định xử dụng phép chiếu trực giao hay phép chiếu phối cảnh điều này cho phép thay đổi giữa 2 phép chiếu để ta thấy được sự khác biệt giữa chúng

```
bool perspective = true;

void display()
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
```

Nếu như bạn muốn di chuyển vị trí của camera (góc nhìn) bạn sẽ phải sửa đổi ma trận chiếu. Điều này là khá phức tạp, có cách đơn giản hơn là ta sử dụng chức năng **gluLookAtf** của thư viện GLU|ES. Tương tự chức năng trong UG là **gluLookAtf**

Chức năng này sẽ đưa ra 9 tham số điều này bao gồm 3 tọa độ hoặc vectors, đầu tiên bạn phải xác định nơi đặt camera, thứ 2 là xác định điểm mà bạn muốn camera được trỏ đến cuối cùng là chỉ rõ việc chuẩn hóa trên vector. Thường sử dụng (0, 1, 0) cho vector này

Đoạn code dưới đây thể hiện nơi đặt camera cách 2 đơn vị từ gốc và nhìn về phía gốc.

```
      gluLookAtf(
            0.0f, 0.0f, 2.0f,
            0.0f, 0.0f, 0.0f,
            0.0f, 1.0f, 0.0f);
```

Tiếp theo là đoạn code để vẽ 3 hình vuông, mỗi hình sẽ được xuất hiện ở phía sau và dịch sang bên trai của hình phía trước, thay vì tạo ra 1 mảng vertex cho hình vuông chúng tôi sử dụng chức năng **ugSolidCubef** của thư viện UG, chức năng này vẽ ra một hình lập phương ở tọa độ (0, 0, 0). Một số các chức năng khác tương tự:

ugSolidBox(GLfloat Width, GLfloat Depth, GLfloat Height);

ugSolidConef(GLfloat base, GLfloat height, GLint slices, GLint stacks);

ugSolidCubef(GLfloat size);

ugSolidDisk(GLfloat inner_radius, GLfloat outer_radius, GLshort rings, GLshort slices);

ugSolidSpheref(GLfloat radius, GLint slices, GLint stacks);

ugSolidTorusf(GLfloat ir, GLfloat or, GLint sides, GLint rings);

ugSolidTube(GLfloat radius, GLfloat height, GLshort stacks, GLshort slices);

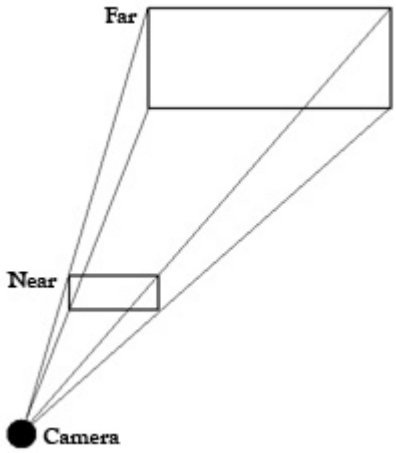
```
        glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
        glTranslatef(-0.25f, 0.0f, -1.0f);
        ugSolidCubef(0.5f);

        glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
        glTranslatef(-0.25f, 0.0f, -1.0f);
        ugSolidCubef(0.5f);

        glFlush();
        glutSwapBuffers();
 }
```

Chức năng reshape ban đầu của chúng tôi vân giữ nguyên

```
void reshape(int width, int height)
{
      w = width;
      h = height;

      if (!height)
            height = 1;

      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();

      glViewport(0, 0, width, height);
```

Giông sử dụng glOrthof để tạo ra hình chiếu trực giao. **glFrustumf** được sử dụng để tạo ra hình chiếu phối cảnh, các tham số cũng giống như hàm **glOrthof** như trái, phải, dưới, trên, gần, xa.

Nó sẽ tạo ra một góc nhìn nhỏ hơn đối với ảnh ở vị trí thấp hơn



Như các bạn đã thấy, chức năng này không trực quan. Một chức năng khác, **gluPerspectivef** đã được tạo ra để xử lí điều này. Cũng giống như chức năng **gluLookAtf**, thư viên UG tương ứng là chức năng ugluPerspectivef và nó có các tham số sau:

GLfloat fovy: điều này chỉ ra phạm vi của góc nhìn. Một góc 90 độ nghĩa là bạn có thể nhìn thấy được mọi thứ ở bên trái và bên phải của bạn, nhưng đây không phải là cách thức mà con người nhìn thấy vật, tôi sử dụng góc 45 độ để chính xác hơn.

GLfloat aspect: điều này chỉ ra tỉ lệ bạn mong muốn, nó thường được chỉ định như là chiều rông chia cho chiều cao của cửa sổ.

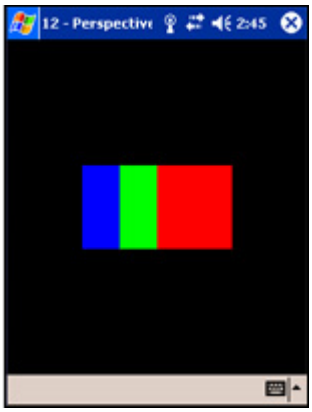
GLfloat n & **GLfloat** f: điều này xác định khoảng cách gần hay xa của (This specifies the near and far clipping planes as normal.)

Đoạn code dưới đây thiết lập góc nhìn theo chiếu phối cảnh hay chiếu trực giao tùy thuộc vào giá trị của biến perspective.

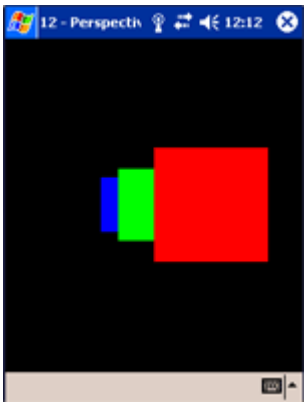
```
        if (perspective)
            gluPerspectivef(45.0f, 1.0f * width / height, 1.0f, 100.0f);
        else
            glOrthof(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 20.0f);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }
```

Bây giờ bạn có thể lựa chọn nhìn theo chiếu phối cảnh hay chiếu trực giao

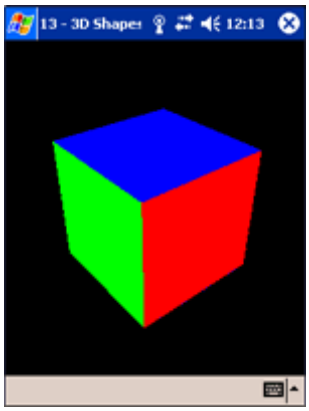
Phép chiếu trực giao



Phép chiếu phối cảnh



3.9 Hình khối (Solid Shapes)



Bây giờ chúng ta đã có khả năng xử lí chiều sâu, và có thể hiển thị đối tượng theo hình chiếu phối cảnh, chúng ta có thể tạo ra một đối tượng 3D

Nội dung của hàm main.cpp

Dưới đây chúng tôi tạo một mảng các đỉnh để tạo ra hình hộp, nhận thấy rằng chúng tôi không tạo ra hình hộp bằng cách sử dụng các giải tam giác liên tục, chúng tôi tạo ra nó bằng cách tạo ra các bề mặt riêng biệt.

```
GLfloat box[] = {
            // FRONT
            -0.5f, -0.5f,  0.5f,
             0.5f, -0.5f,  0.5f,
            -0.5f,  0.5f,  0.5f,
             0.5f,  0.5f,  0.5f,
            // BACK
            -0.5f, -0.5f, -0.5f,
            -0.5f,  0.5f, -0.5f,
             0.5f, -0.5f, -0.5f,
             0.5f,  0.5f, -0.5f,
            // LEFT
            -0.5f, -0.5f,  0.5f,
            -0.5f,  0.5f,  0.5f,
            -0.5f, -0.5f, -0.5f,
            -0.5f,  0.5f, -0.5f,
            // RIGHT
             0.5f, -0.5f, -0.5f,
             0.5f,  0.5f, -0.5f,
             0.5f, -0.5f,  0.5f,
             0.5f,  0.5f,  0.5f,
            // TOP
            -0.5f,  0.5f,  0.5f,
             0.5f,  0.5f,  0.5f,
             -0.5f,  0.5f, -0.5f,
             0.5f,  0.5f, -0.5f,
            // BOTTOM
            -0.5f, -0.5f,  0.5f,
            -0.5f, -0.5f, -0.5f,
             0.5f, -0.5f,  0.5f,
             0.5f, -0.5f, -0.5f,
};
```

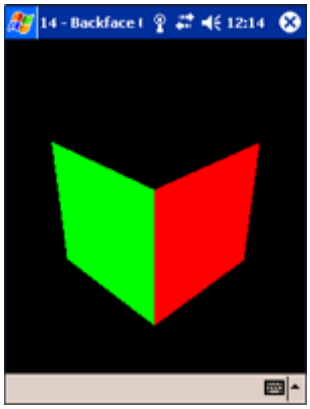
Bước tiếp theo là thiết lập màn hình và xoay như bình thường

```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  gluLookAtf(
                    0.0f, 0.0f, 3.0f,
                    0.0f, 0.0f, 0.0f,
                    0.0f, 1.0f, 0.0f);

  glRotatef(xrot, 1.0f, 0.0f, 0.0f);
  glRotatef(yrot, 0.0f, 1.0f, 0.0f);
```

Chúng tôi muốn vẽ 2 mặt đối diện có màu giống nhau vì vậy nên ta vẽ 2 mặt cùng một lúc.

```
    glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);
    glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
    glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);
    glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);

    glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
    glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);
    glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);

    glFlush();
    glutSwapBuffers();
}
```

3.10 Bộ lọc mặt sau (Backface Culling)



Trong phần hướng dẫn thứ 3.8 ta nhận thấy các hình sau khi quay mặt sau của chúng cũng được đưa ra, khi tạo ra đối tượng 3D như hình hộp trong hướng dẫn trước, chúng tôi không cần mặt sau của các mặt được hiển thị

Một kĩ thuật được gọi là Backface Culling được sử dụng để ngăn chặn các mặt trong của hình được đưa ra. Điều này có thể tiết kiệm được thời gian để vẽ và bộ nhớ.

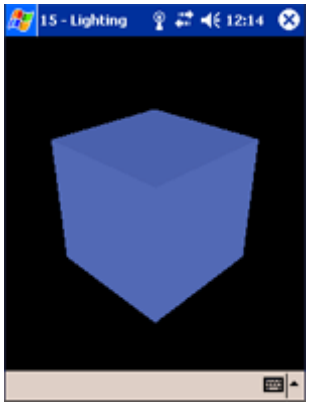
Nội dung của hàm main.cpp

Bước đầu tiên mà chúng ta cần phải thực hiện để kích hoạt chế độ backface culling bằng cách thêm các đoạn mã dưới đây vào hàm init đẻ kích hoạt chức năng backface culling chúng tôi phải sử dụng cờ GL_CULL_FACE điều này sẽ làm cho tất cả các mặt sau của hình không bị đưa ra.

Bạn có thể hỏi là làm thế nào để có thế xác định được mặt sau của hình? Khi bạn vẽ các hình, bạn chỉ định các đỉnh trong mảng theo hướng chiều kim đồng hồ vì vậy nếu bạn đẻ ý trong ma trận mà chúng tôi đưa ra, tất cả các hình đã được chỉ định đưa ra đỉnh theo hướng cùng chiều kim đồng hồ

```
glEnable(GL_CULL_FACE);
```

3.11 Ánh sáng (Lighting)



Bước đầu tiên ta cần thực hiện là kích hoạt backface culling như trong hướng dẫn trước, phần này sẽ hướng dân làm thế nào để thêm ánh sáng vào cảnh của bạn. Điều này làm tăng tính chân thực và cách nhìn của bạn.

Có một số loại ánh sáng có thể được thêm vào hình của bạn:

Ambient Light: Ánh sáng bao xung quanh, nó không đến từ bất kì một hướng nào cụ thể, khi ánh sáng bao xung quanh một bề mặt ánh sáng sẽ được phản xạ theo nhiều hướng.

Diffuse Light: Ánh sáng khuếch tán, nó đến từ một hướng, ánh sáng khuếch tán tương tự như anh sáng bao quanh nó cũng được phản xạ theo nhiều hướng.

Specular Light: Ánh sáng phản chiếu, cũng giống như ánh sáng khuếch tán nhưng nó được phản xạ theo một hướng, như là bạn có thể thấy ánh sáng nổi bật trên bề mặt trước.

Emissive Light: Ánh sáng tỏa, ánh sáng này đến từ một đối tượng cụ thể, các đối tượng cso thể giảm lượng ánh sáng nhưng nó không thể phản chiếu ra bất kì bề mặt ngoài nào.

Không chỉ có thể thắp sáng các thuộc tính mà bạn chỉ định, bạn có thể chỉ định các bề mặt phản ứng như thế nào với ánh sáng

Pháp tuyến là một vector vuông góc với một bề mặt. nó được sử dụng trong việc tính toán ánh sáng bạn cần phải xác định một pháp tuyến cho mọi đa giác được vẽ nếu bạn muốn nó bị ảnh hưởng bởi nguồn sáng.

<u>Nội dung của hàm main.cpp</u>

Dưới đấy tôi sẽ tạo ra 2 mảng màu cho ánh sáng bao quanh và ánh sáng khuếch tán. Đây sẽ là màu sắc của ánh sáng nguồn.

```
float lightAmbient[] = { 0.2f, 0.3f, 0.6f, 1.0f };
float lightDiffuse[] = { 0.2f, 0.3f, 0.6f, 1.0f };
```

Tiếp theo ta sẽ tạo ra 1 mảng chất liệu, một ánh sáng bao quanh và một ánh sáng khuêch tán cho nguồn

Về bản chất điều này làm tăng giá trị của ánh sáng bởi các giá trị của chất liệu nó làm cho màu sắc phản chiếu lên các bề mặt bị mất. Các mảng ở bề mặt dưới mất đến 40% ánh sáng , mỗi giá trị tượng trưng cho màu mà nó phản xa.

```
float matAmbient[] = { 0.6f, 0.6f, 0.6f, 1.0f };
float matDiffuse[] = { 0.6f, 0.6f, 0.6f, 1.0f };

void init()
{
```

Bước đầu tiên phải bật cờ GL_LIGHTING trong hàm glEnable điều này cho phép sử dụng ánh sáng trong OpenGL

```
    glEnable(GL_LIGHTING);
```

OpenGL cho phép bạn có tối đa 8 nguồn sáng từ bất kì điểm nào để kích hoạt được các nguồn sáng này bạn phải bật cờ GL_LIGHTX trong hàm glEnable, X là giá trị từ 0 đến 7.

```
    glEnable(GL_LIGHT0);
```

Xác định các thông số chất liệu cho các mô hình chiếu sáng, thông qua các chức năng **glMaterialfv** và **glMaterialf** cùng với 3 tham số.

- Tham số thứ nhất là cờ `GL_FRONT_AND_BACK`
- Tham số thứ hai dùng để xác định loại nguồn sáng mà bạn muốn sử dụng như GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION và GL_AMBIENT_AND_DIFFUSE
- Tham số cuối cùng là một mảng hoặc một giá trị

```
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, matAmbient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, matDiffuse);
```

Giống như việc thiết lập chất liệu, ánh sáng cũng được thiết lập như vậy, điều này được thực hiện bằng cách sử dụng chức năng **glLightfv** và **glLightf**

```
    glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);
```

Phần còn lại của hàm **Init** vẫn được giữ nguyên

```
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LEQUAL);

        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
        glClearDepthf(1.0f);

        glVertexPointer(3, GL_FLOAT, 0, box);
        glEnableClientState(GL_VERTEX_ARRAY);

        glEnable(GL_CULL_FACE);
        glShadeModel(GL_SMOOTH);
}
```

Phần đầu của hàm **display** vẫn được giữ nguyên

```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  gluLookAtf(
          0.0f, 0.0f, 3.0f,
          0.0f, 0.0f, 0.0f,
          0.0f, 1.0f, 0.0f);

  glRotatef(xrot, 1.0f, 0.0f, 0.0f);
  glRotatef(yrot, 0.0f, 1.0f, 0.0f);
```

Ở phần trên chúng ta đã nói về pháp tuyến, các pháp tuyến này cần vuông góc với bề mặt, bởi vậy bề mặt phía trước có một vector pháp tuyến (0,0,1), phía sau là (0,0,-1). Độ dài 2 vector này là 1

Các pháp tuyến được xác định bằng hàm **glNormal3**f và nó được gọi trước khi vẽ hình, hàm này có 3 tham số float.

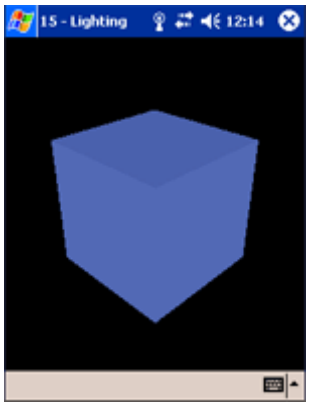
```
  // FRONT AND BACK
  glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
  glNormal3f(0.0f, 0.0f, 1.0f);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
  glNormal3f(0.0f, 0.0f, -1.0f);
  glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);
```

Điều này cũng được thực hiện cho phía trên, phía dưới và các mặt

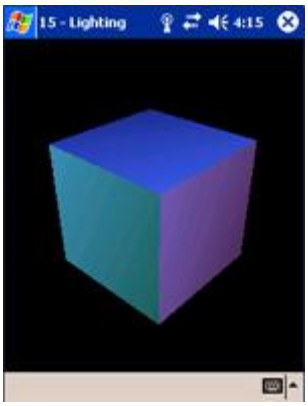
```
    // LEFT AND RIGHT
    glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
    glNormal3f(-1.0f, 0.0f, 0.0f);
    glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);
    glNormal3f(1.0f, 0.0f, 0.0f);
    glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);

    // TOP AND BOTTOM
    glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
    glNormal3f(0.0f, 1.0f, 0.0f);
    glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);
    glNormal3f(0.0f, -1.0f, 0.0f);
    glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);

    glFlush();
    glutSwapBuffers();
}
```

Việc bật và tắt việc gọi đến color tracking thông qua cờ GL_COLOR_MATERIAL trong hàm glEnable, **Color tracking** nó sẽ tự động đặt thuộc tính chất liệu theo lời gọi đến **glColor4f ,** việc làm này sẽ làm cho các mặt phản xạ ánh sáng với màu sắc khác nhau

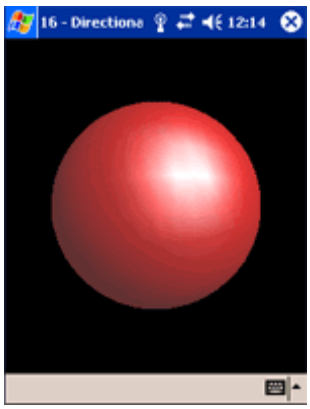
Normal Lighting



Color Tracking



3.12 Định hướng ánh sáng (Directional Lighting)



Trong phần trước ta đã thêm ánh sáng vào cảnh, nhưng ánh sáng không đến từ một hướng cụ thể

Trong phần này ta sẽ giải quyết việc định hướng nguồn sáng, điều này sẽ cho phép ta sự dụng lợi ích của khuếch tán và phản chiếu ánh sáng.

Nội dung của hàm main.cpp

Một lần nữa ta lại tạo các mảng ánh sáng cho các đặc tính ánh sáng, chúng ta thêm mảng specular.

```
float lightAmbient[] = { 0.2f, 0.0f, 0.0f, 1.0f };
float lightDiffuse[] = { 0.5f, 0.0f, 0.0f, 1.0f };
float lightSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

Một mảng specular cho chất liệu cũng là cần thiết

```
float matAmbient[] = { 1.0f, 1.0f, 1.0f, 1.0f };
float matDiffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f };
float matSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

Vì đây là định hướng nguồn sáng nên chúng ta cần phải biết vị trí của ánh sáng và hướng của nó. Đoạn code dưới đây sẽ tạo ra 2 mảng để đặt ánh sáng trong không gian phía bên phải của quả bóng, nó sẽ hướng về phía gốc nên cần 1 vector chỉ phương hướng (-2, -2, -3).

```
float lightPosition[] = { 2.0f, 2.0f, 3.0f, 0.0f };
float lightDirection[] = { -2.0f, -2.0f, -3.0f };
```

Nguồn sáng sẽ được bật cùng với những ánh sáng đầu tiên

```
void init()
{
	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);
```

Tất cả các thuộc tính cho chất liệu bao gồm cả giá trị specular

```
	glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, matAmbient);
	glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, matDiffuse);
	glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, matSpecular);
```

Một thiết lập khác bằng cách sử dụng chức năng **glMaterialf** với đặc tính GL_SHININESS. Giá trị shininess trong khoảng từ 0 đến 128. Điều này chỉ tập chung làm thế nào để specular sẽ được tô sáng.

```
	glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 20.0f);
```

Bước tiếp theo là thiết lập thuộc tính ánh sáng

```
	glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);
	glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);
	glLightfv(GL_LIGHT0, GL_SPECULAR, lightSpecular);
```

Thiết lập vị trí và định hướng ánh sáng thông qua cờ GL_POSITION và GL_SPOT_DIRECTION trong hàm **glLightfv**

```
	glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
	glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, lightDirection);
```

Một cờ khác GL_SPOT_CUTOFF được sử dụng để xác định kích cỡ của nguồn sáng

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 1.2f);
```

Tiếp theo ta sẽ sử dụng cờ GL_SPOT_EXPONENT dùng để xác định cách thức tập trung của nguồn sáng như cờ GL_SHININESS với giá trị từ 0 đến 128

```
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 20.0f);
```

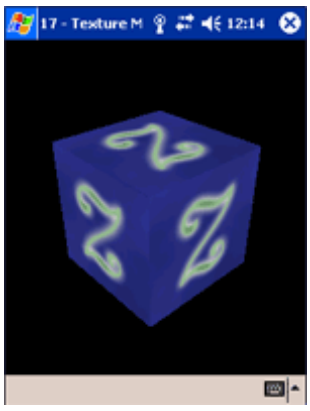
Phần còn lại của hàm Init vẫn được giữ nguyên, hiện có 3 cờ được sử dụng trong hàm glLightf là GL_CONSTANT_ATTENUATION(1), GL_LINEAR_ATTENUATION(0) và GL_QUADRATIC_ATTENUATION với các giá trị hiển thị mặc định trong dấu (). Cường độ ánh sáng bị suy yếu khi bạn di chuyển mảng ra xa khỏi nguồn sáng.

```
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepthf(1.0f);

    glEnable(GL_CULL_FACE);
    glShadeModel(GL_SMOOTH);
}
```

Màn hình hiển thị sử dụng chức năng glutSolidSphere / ugSolidSpheref để tạo ra một hình cầu, hình cầu này được tạo ra với 24 stacks và 24 slices, đây là thành phần ngang va dọc của hình cầu.

Bạn có thể tự hỏi nơi mà chúng tôi xác định pháp tuyến, chức năng shape của thư viện UG / GLUT|ES sẽ tự động tạo ra một mảng vector pháp tuyến và sử dụng, mọi pháp tuyến sẽ được tính toán trong thư viện này.

3.13 Dán chất liệu (Texture Mapping)



Sau khi thêm ánh sáng chắc hẳn bạn vẫn chưa hài lòng với những gì hiển thị của đối tượng.

Phần hướng dẫn này sẽ hướng dẫn bạn làm thế nào để thêm chất liệu vào đối tượng và nó được gọi là texture mapping.

Bước đầu tiên của texture mapping là nạp các file chất liệu từ bên ngoài. Các file này có thể có các đuôi như bmp, jpg, gif, png

v.v…Trong phần hướng dẫn này ta chỉ làm việc với file bmp bởi vì nó dễ nạp vào nhất. OpenGL cũng lưu ý rằng cần phải làm việc với những ảnh có kích thước là lũy thừa của 2 như 64x64, 128x128, 256x128 v.v…

Nội dung của hàm main.cpp

Tất cả các chất liệu đều có một định dạng cụ thể. Điều này được thể hiện như là một unsigned integer, chúng ta sẽ tạo ra một mảng để chứa đủ một chất liệu

```
GLuint texture[1];
```

Sau khi tải chất liệu vào chúng ta phải chỉ rõ chất liệu sẽ xuất hiện như thế nào trên đối tượng. Điều này được thực hiện bằng lời gọi hàm texture coordinates

Texture coordinates có tọa độ trong khoảng từ 0 đến 1, tọa đô (0. 0) là phía dưới bên trái của chất liệu và (0, 1) là phía trên bên phải.

Đoạn code dưới đây tạo ra 1 mảng được sử dụng để lưu trữ texture coordinates

```
GLfloat texCoords[] = {
      // FRONT
       0.0f, 0.0f,
       1.0f, 0.0f,
       0.0f, 1.0f,
       1.0f, 1.0f,
      // BACK
       1.0f, 0.0f,
       1.0f, 1.0f,
       0.0f, 0.0f,
       0.0f, 1.0f,
      // LEFT
       1.0f, 0.0f,
       1.0f, 1.0f,
       0.0f, 0.0f,
       0.0f, 1.0f,
      // RIGHT
       1.0f, 0.0f,
       1.0f, 1.0f,
       0.0f, 0.0f,
       0.0f, 1.0f,
      // TOP
       0.0f, 0.0f,
       1.0f, 0.0f,
       0.0f, 1.0f,
       1.0f, 1.0f,
      // BOTTOM
       1.0f, 0.0f,
       1.0f, 1.0f,
       0.0f, 0.0f,
       0.0f, 1.0f
};
```

Những chức năng dưới đây dùng để tải 1 file bitmap, trước khi bạn nạp một bitmap bạn cần phải hiểu cấu trúc của nó. Đầu tiên là tiêu đề của bitmap, nó chứa các thuộc tính như kiểu tệp tin và nơi mà bitmap được đặt. Đây là những thông tin cần thiết để nạp vào một BITMAPFILEHEADER. Sau tiêu đề bạn sẽ tìm thấy được một số các thông tin như chiều rông, chiều cao, và các bits trên mỗi điểm ảnh của ảnh. Các thông tin này sẽ được nạp vào BITMAPFILEHEADER. Cuối cùng là dữ liệu ảnh hiện thời sau 2 tiêu đề

Hàm dưới đây sẽ cần 2 tham số. Tham số đầu tiên để chỉ rõ tên cảu ảnh bitmap cần nạp vào, nó sẽ được tìm trong thư mục hiện thời. tham số thứ hai là một con trỏ trỏ tới BITMAPINFOHEADER

```
unsigned char *loadBMP(char *filename, BITMAPINFOHEADER *bmpInfo)
{
```

Chúng tôi sẽ tạo ra một số biến. đầu tiên là một con trỏ trỏ đến cấu trúc FILE để mở tệp tin này

```
    FILE *file;
```

Như đã nói ở trên chúng ta phải nạp các tiêu đề của tệp tin, một BITMAPFILEHEADER được sử dụng.

```
    BITMAPFILEHEADER bmpFile;
```

Dữ liệu ảnh có thể được đại diện bởi một số của unsigned char's. chúng ta sẽ tạo ra 1 mảng để chứa dữ liệu này.

```
    unsigned char *bmpImage = NULL;
```

Ảnh bitmap lưu trữ dữ liệu điểm ảnh theo định dạng BGR. Chúng tôi không muốn điều này khi chúng tôi làm việc theo giá trị RGB. Biến tmpRGB dưới đây sẽ giúp chúng tôi giải quyết vấn đề này.

```
    unsigned char tmpRGB;
```

Một rủi ro về việc xác định vị trí làm việc hiện thời. biến path sẽ chứa đường dẫn của thư mục hiện thời, và biến fullPath sẽ chứa đường

```
    TCHAR path[256];
    char fullPath[256];
```

Bước đầu tiên trong việc xác định thư mục hiện thời là lời gọi đến hàm GetModuleFileName, hàm này có 3 tham số. Tham số đầu tiên là xác định modun bạn đang xem, nếu được phép thì Null, bạn sẽ sử dụng modun hiện tại. Tham số thứ hai xác định đường dẫn được lưu trữ và tham số thứ ba xác định số lượng kí tự tối đa để nạp vào

```
GetModuleFileName(NULL, path, 256);
```

Bước đầu tiên để tìm các vị trí cuối cùng của kí tự '\'. Điều này được thực hiện bởi lời gọi hàm wcsrchr

```
TCHAR *pos = wcsrchr(path, '\\');
```

Thay vì loại bỏ các thành phần còn lại của chuỗi chúng ta chỉ cần đặt kí tự Null sau vị trí được tìm thấy.

```
*(pos + 1) = '\0';
```

Bây giờ chúng ta phải chuyển đổi đường dẫn này đến multibyte character. Điều này đạt được bằng cách sử dụng chức năng wcstombs.

```
wcstombs(fullPath, path, 256);
```

Bước cuối cùng trong việc xác định vị trí của bitmap là kết nối đường dẫn cũng với tên file thông qua chức năng **strcat**

```
strcat(fullPath, filename);
```

Bây giờ ta đã có vị trí của bitmap và có thể mở nó theo chế độ nhị phân

```
file = fopen(fullPath,"rb");
```

Nếu tệp tin không được tìm thấy chúng ta sẽ hiện thị một thông báo lỗi

```
if (!file)
{
    MessageBox(NULL, L"Can't Find Bitmap", L"Error", MB_OK);
    return NULL;
}
```

Các BITMAPFILEHEADER được đọc vào trong cấu trúc

```
fread(&bmpFile,sizeof(BITMAPFILEHEADER),1,file);
```

Mỗi ảnh bitmap có một ID là 0x4D42, giá trị này được lưu trữ trong các biến bfType của BITMAPFILEHEADER. Nếu ID không được tìm thấy thì sẽ ngừng việc tải ảnh.

```
if (bmpFile.bfType != 0x4D42)
{
	MessageBox(NULL, L"Incorrect texture type", L"Error", MB_OK);
	fclose(file);
	return NULL;
}
```

Bước tiếp theo là để nạp vào cấu trúc BITMAPINFOHEADER

```
fread(bmpInfo,sizeof(BITMAPINFOHEADER),1,file);
```

BITMAPFILEHEADER chứa một thuộc tính bfOffBits, nó dùng để xác định số bits thực tế trên ảnh. Vì vậy chúng tôi di chuyển con trỏ từ đầu tệp tin (SEEK_SET) đến phần bắt đầu của dữ liệu ảnh.

```
fseek(file,bmpFile.bfOffBits,SEEK_SET);
```

Bộ nhớ được cấp phát cho dữ liệu ảnh. Kích cỡ của ảnh được lưu trữ ở trong cấu trúc BITMAPINFOHEADER

```
bmpImage = new unsigned char[bmpInfo->biSizeImage];

if (!bmpImage)
{
	MessageBox(NULL, L"Out of Memory", L"Error", MB_OK);
	delete[] bmpImage;
	fclose(file);
	return NULL;
}
```

Sau khi bộ nhớ được phân bổ, chúng ta cần phải nạp dữ liệu ảnh từ tệp tin từng bit cùng thời điểm.

```
fread(bmpImage,1,bmpInfo->biSizeImage,file);

if (!bmpImage)
{
	MessageBox(NULL, L"Error reading bitmap", L"Error", MB_OK);
	fclose(file);
	return NULL;
}
```

Nhớ rằng chúng ta nói ảnh bitmap lưu trữ các điểm ảnh theo định dạng BGR. Vì vậy chúng ta phải chuyển đổi lại sang định dạng RGB.

```
for (unsigned int i = 0; i < bmpInfo->biSizeImage; i+=3)
{
	tmpRGB = bmpImage[i];
	bmpImage[i] = bmpImage[i+2];
	bmpImage[i+2] = tmpRGB;
}
```

Bước cuối cùng là đóng tệp tin và sử dụng con trỏ để trỏ đến dữ liệu ảnh

```
        fclose(file);

        return bmpImage;
}
```

Bây giờ chúng ta đã có 1 chức năng tải các ảnh bipmaps. Thay vì đặt các lệnh nạp ảnh bitmap và dán chất liệu trong hàm **Init**. Ta sẽ đặt nó trong hàm **loadTextures.**

```
bool loadTextures()
{
```

Như đã nói ở phía trên chúng ta phải tạo ra một BITMAPINFOHEADER để lưu trữ giữ liệu ảnh. Chúng ta cũng có thể tạo ra một con trỏ để trỏ đến dữ liệu hình ảnh.

```
        BITMAPINFOHEADER info;
        unsigned char *bitmap = NULL;
```

Bước kế tiếp là nạp ảnh bipmap để sử dụng cho các chức năng ở phía trên

```
        bitmap = loadBMP("zeus.bmp", &info);

        if (!bitmap)
                return false;
```

Như đã nói ở phía trên, mọi chất liệu trong OpenGL được gọi thông qua tên của chất liệu. Để tạo ra các định danh, chúng ta cần sử dụng chức năng **glGenTextures**, chức năng này có hai tham số, tham số đầu tiên để xác định có bao nhiêu chất liệu mà bạn muốn tạo ra. Tham số thứ hai là một con trỏ trỏ tới mảng unsigned integers. Điều này sẽ giúp bạn tạo ra các tên cho chất liệu.

```
        glGenTextures(1, texture);
```

Bây giờ thì tên của chất liệu đã được tạo ra, bạn phải lựa chọn chất liệu mà bạn muốn thiết lập trong hiện tại. Điều này thực hiện được bằng cách sử dụng hàm **glBindTexture. Hàm này có hai tham số, tham số đầu tiên là** GL_TEXTURE_2D và tham số thứ hai chấp nhận chất liệu mà bạn muốn lựa chọn.

```
        glBindTexture(GL_TEXTURE_2D, texture[0]);
```

Sau khi lựa chọn chất liệu, chúng ta phải thiết lập các thuộc tính cho nó. Chúng ta cần phải xác định chất liệu là các kết cấu 2D và các thuộc tính nó

có. Điều này được thực hiện qua hàm **glTexImage2D** hàm này có một số các tham số:

- GLenum **target:** điều này xác định đích của chất liệu là GL_TEXTURE_2D, OpenGL ES không hỗ trợ chất liệu 1D hoặc 3D.
- GLint **level:** dùng để xác định mức độ chi tiết, 0 là cấp độ hình ảnh cơ bản, nó chỉ được sử dụng cho mipmaps nơi có các chất liệu khác nhau tùy thuộc vào khoảng cách của chất liệu đến người xem.
- GLint **internalFormat**: điều này xác định màu sắc cho các thành phần bên trong chất liệu. Nó có thể là GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE hoặc GL_LUMINANCE_ALPHA nhưng chúng ta thường chỉ sử dụng 2 cờ GL_RGB hoặc GL_RGBA. Cờ GL_RGBA chỉ được sử dụng khi bạn sử dụng một ảnh chứa giá trị anpha như tệp tin tga.
- GLsizei **width** & GLsizei **height**: dùng để xác định chiều rộng và chiều cao của ảnh. Điều này có thể lấy từ cấu truc BITMAPINFOHEADER.
- GLint **border**: điều này chỉ ra độ rộng của đường biên. Nó có giá trị là 0.
- GLenum **format**: điều này được đưa ra cùng giá trị như là tham số internalFormat.
- GLenum **type**: điều này dùng để xác định kiểu dữ liệu đang được lưu trữ ảnh ví dụ như GL_UNSIGNED_BYTE và GL_UNSIGNED_SHORT.
- const GLvoid ***pixels**: điều này chỉ ra nơi mà ảnh được lưu trữ.

Nếu bạn chỉ muốn sử dụng một phần của hình ảnh, chức năng **glTexSubImage2D** có thể được sử dụng. Các tham số của nó giống nhau ngoại trừ tham số **internalFormat**, có hai tham số khác là GLint **xoffset** và GLint **yoffset**, điều này để chỉ ra khoảng cách x, y cho ảnh.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info.biWidth,
    info.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
    bitmap);
```

Để thiết lập các thuộc tính khác cho chất liệu, chức năng **glTexParameterf** được sử dụng. Điều này sẽ được thảo luận trong phần sau, bây giờ tất cả những gì bạn cần biết là nó cần 3 tham số. Tham số đầu tiên là GL_TEXTURE_2D.Tham số thứ hai GL_TEXTURE_MIN_FILTER hoặc GL_TEXTURE_MAG_FILTER, tham số thứ ba xác định cái mà bạn muốn thiết lập cho thuộc tính - GL_LINEAR.

```
        glTexParameterf(GL_TEXTURE_2D,
            GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D,
            GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Các chất liệu đã được thiết lập, chúng ta phải giải phóng bộ nhớ cấp phát cho bitmap.

```
        delete[] bitmap;

        return true;
}
```

Trong hàm **Init** chúng tôi sẽ thêm lời gọi hàm đến **loadTextures**.

```
bool init()
{
        if (!loadTextures())
        {
            MessageBox(NULL, L"Error loading textures", L"Error", MB_OK);
            return false;
        }
```

Đê bật chức năng texture mapping, chúng ta cần phải sử dụng cờ GL_TEXTURE_2D trong hàm **glEnable**

```
        glEnable(GL_TEXTURE_2D);

        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        .
        .
```

Như chúng ta làm trên vertices. Chúng ta cần phải xác định vị trí của texture coordinates điều này được thực hiện theo cùng một cách như với các vertices ngoại trừ bây giờ chúng ta sử dụng chức năng glTexCoordPointer. Chú ý rằng giá trị 2 được sử dụng cho tham số đầu tiên bởi vì mỗi texture coordinate chỉ có 2 giá trị.

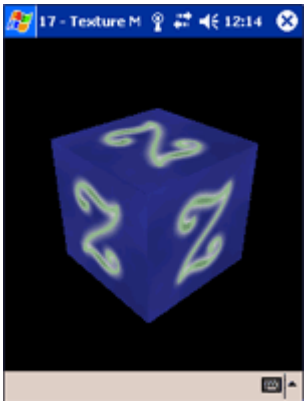
```
        .
        .
        glVertexPointer(3, GL_FLOAT, 0, box);
        glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
```

Bước cuối cùng là cho phép bật mảng texture coordinate thông qua cờ GL_TEXTURE_COORD_ARRAY trong hàm glEnableClientState

```
        glEnableClientState(GL_VERTEX_ARRAY);
        glEnableClientState(GL_TEXTURE_COORD_ARRAY);
        .
        .
}
```

Một hàm menu dùng để bât và tắt ánh sáng. Khi tắt ánh sáng màu sắc bạn sử dụng khi tạo ra các mặt sẽ được kết hợp vào trong chất liệu, bạn có thể nhìn thấy ở hình dưới. Khi ánh sáng được bật ánh sáng phản xạ sẽ phạn xạ một số lượng ánh sáng bằng nhau cho tất cả các màu do đó chất liệu sẽ được xuất hiện như bình thường.

Lighting Enabled



Lighting Disabled

```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  gluLookAtf(
          0.0f, 0.0f, 4.0f,
          0.0f, 0.0f, 0.0f,
          0.0f, 1.0f, 0.0f);

  glRotatef(xrot, 1.0f, 0.0f, 0.0f);
  glRotatef(yrot, 0.0f, 1.0f, 0.0f);

  ugSolidSpheref(1.0f, 24, 24);

  glFlush();
  glutSwapBuffers();
}
```

Sau khi chạy chương trình bạn sẽ thấy một quả bóng màu đỏ cùng với ánh sáng phản chiếu ở phía trên bên phải của quả bóng, bây giờ bạn có thể tự thêm các định hướng nguồn sáng vào chương trình của bạn.

3.14 Hàm chất liệu (Texture Functions)

Trong phần trước chúng ta đã biết cách làm thế nào để nạp một ảnh bitmap và hiển thị nó như là chất liệu của đối tượng. Bạn đã được giới thiệu ngắn gọn về hàm glTexParameterf.

Phần này sẽ thảo luận tiếp về chức năng này với nhiều khía cạnh khác nhau.

Texture Filters: bộ lọc chất liệu cho phép chất liệu được hiện thị cùng với các chất lượng khác nhau

Repeating and Clamping: chất liệu có thể được lặp đi lặp lại trên đối tượng.

Mipmaps: Mipmaps là tạo ra thêm chất liệu của cùng một hình ảnh. Nếu bạn có một bức ảnh cỡ 64x64. Hình ảnh thêm được tạo ra (32x32, 16x16, ..., 1x1). Chất liệu chính xác sẽ được hiện thị tùy thuộc vào khoảng cách của mảng đối tượng. rõ ràng các mảng đối tượng ở xa hơn thì sử dụng lớp chất liệu nhỏ hơn. Điều này có thể tiết kiệm được thời gian xử lý.

Nội dung của hàm main.cpp

Chúng ta sẽ tạo ra 4 chất liệu khác nhau trong hướng dẫn này. Chúng tôi cũng muốn theo dõi chất liệu bằng cách sử dụng biến filter.

```
GLuint texture[4];

short filter = 0;
```

Tọa độ chất liệu chúng ta vẫn giữ nguyên ngoại trừ các bề mặt bên. Lưu thông tin mà chúng tôi cho rằng có giá trị từ 0 đến 1. Chúng ta có thể sử dụng giá trị lớn hơn 1. Nhưng điều này sẽ gây ra lạp lại chất liệu hay dừng khi đạt được 1. điều này sẽ được giải thích rõ hơn ở phần dưới.

```
GLfloat texCoords[] = {
      // FRONT
       0.0f, 0.0f,
       1.0f, 0.0f,
       0.0f, 1.0f,
       1.0f, 1.0f,
      // BACK
       1.0f, 0.0f,
       1.0f, 1.0f,
       0.0f, 0.0f,
       0.0f, 1.0f,
      // LEFT
       2.0f, 0.0f,
       2.0f, 2.0f,
       0.0f, 0.0f,
       0.0f, 2.0f,
      // RIGHT
       2.0f, 0.0f,
       2.0f, 2.0f,
       0.0f, 0.0f,
       0.0f, 2.0f,
      // TOP
       0.0f, 0.0f,
       1.0f, 0.0f,
       0.0f, 1.0f,
       1.0f, 1.0f,
      // BOTTOM
       1.0f, 0.0f,
       1.0f, 1.0f,
       0.0f, 0.0f,
       0.0f, 1.0f
};
```

Chúng ta xây dựng hàm loadTextures và nạp file .bmp như phần trước.

```
bool loadTextures()
{
      BITMAPINFOHEADER info;
      unsigned char *bitmap = NULL;

      bitmap = loadBMP("zeus.bmp", &info);

      if (!bitmap)
            return false;
```

Chúng ta cần phải tạo ra 4 tên chất liệu

```
      glGenTextures(4, texture);
```

Chúng ta lựa chọn chất liệu đầu tiên và thiết lập các thuộc tính như ở hướng dẫn trước. Sự khác biệt ở đây là chúng ta dùng cờ GL_NEAREST trong thuộc tính của bộ lọc thay cho GL_LINEAR. Bộ lọc này nhanh hơn nhưng nhìn không được tốt lắm.

```
      // Texture 1
      glBindTexture(GL_TEXTURE_2D, texture[0]);

      glTexParameterf(GL_TEXTURE_2D,
            GL_TEXTURE_MIN_FILTER, GL_NEAREST);
      glTexParameterf(GL_TEXTURE_2D,
            GL_TEXTURE_MAG_FILTER, GL_NEAREST);
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info.biWidth,
            info.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
            bitmap);
```

Trong chất liệu thứ hai chúng tôi sử dụng GL_LINEAR để tạo ra chất liệu nhìn tốt hơn. Đây là những gì đã được sử dụng trong hướng dẫn trước.

```
// Texture 2
glBindTexture(GL_TEXTURE_2D, texture[1]);

glTexParameterf(GL_TEXTURE_2D,
     GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D,
     GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info.biWidth,
     info.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
     bitmap);
```

Chất liệu thứ ba cũng giống như chất liệu thứ hai ngoại trừ hai thuộc tính mới được thiết lập. Đấy là thuộc tính GL_TEXTURE_WRAP_S và GL_TEXTURE_WRAP_T. Điều này chỉ ra chất liệu làm thế nào để bao bọc theo các hướng ngang và dọc tương ứng.

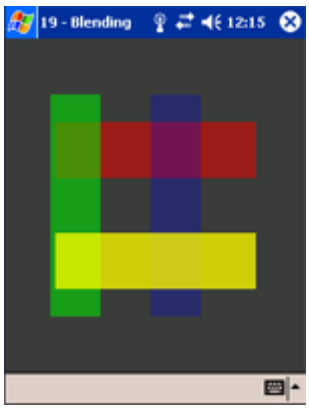
```
// Texture 3
glBindTexture(GL_TEXTURE_2D, texture[2]);

glTexParameterf(GL_TEXTURE_2D,
     GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D,
     GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info.biWidth,
     info.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
     bitmap);
```

Thứ tư là sử dụng các kĩ thuật cao của mipmaps. Mipmaps yêu cầu bộ nhớ nhiều hơn nhưng nó có thể làm cho chương trình của bạn chạy nhanh hơn rất nhiều. Để tự động tạo ra mipmaps bạn có thể thiết lập thuộc tính L_GENERATE_MIPMAP tới GL_TRUE bằng lời gọi hàm glParameterf. GL_TEXTURE_MAG_FILTER của bạn vẫn như cũ nhưng thuộc tính GL_TEXTURE_MIN_FILTER phải thay đổi.

Bộ lọc này phải được thiết lập GL_X_MIPMAP_Y nơi mà X và Y có thể LINEAR hoặc NEAREST. Điều này dùng để xác định chất lượng của chất liệu được hiển thị. Rõ ràng NEAREST nhìn không tốt bằng LINEAR.

```
        // Texture 4
        glBindTexture(GL_TEXTURE_2D, texture[3]);

        glTexParameterf(GL_TEXTURE_2D,
              GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
        glTexParameterf(GL_TEXTURE_2D,
              GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D,
              GL_GENERATE_MIPMAP, GL_TRUE);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, info.biWidth,
              info.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
              bitmap);

        delete[] bitmap;

        glBindTexture(GL_TEXTURE_2D, texture[filter]);

        return true;
}
```

3.15 Pha trộn (Blending)



Hướng dẫn này sẽ giới thiệu về cách pha trộn màu. Việc pha trộn màu sắc rất hữu ích cho các hiệu ứng ví dụ như: thủy tinh, nước, màn hình v.v..

Một phần thiết yếu của việc trộn màu là giá trị alpha mà chúng ta chỉ định cho tất cả màu sắc. Alpha có giá trị bằng 0 thể hiện một bề mặt hoàn toàn trong suốt và giá trị bằng 1 thể hiện một bề mặt mờ đục.

Khi làm việc với sự trộn màu chúng ta phải luôn nhớ đến 2 màu khác nhau. Thứ nhất là giá trị của màu nguồn (giá trị hiện tại thêm vào) và bản ghi của giá trị màu (giá trị tồn tại trong bộ đệm). Màu sắc sẽ được làm tùy thuộc vào giá trị alpha.

Nội dung của hàm main.cpp

Tong phần này ta sẽ thiết lập góc nhìn theo chiếu trực giao với các tham số:

```
        glOrthof(0.0f, 3.0f, 0.0f, 3.0f, -1.0f, 1.0f);
```

Chúng ta sẽ đặt một số lượng hình chữ nhật chồng lên nhau trên màn hình, các đỉnh cho hình chữ nhật được đưa ra ở mảng dưới đây.

```
GLfloat rectangle[] = {
      -1.0f, -0.25f,
       1.0f, -0.25f,
      -1.0f,  0.25f,
       1.0f,  0.25f
};
```

Chúng ta sẽ hiện thị các kết hợp khác nhau của trộn màu. Những biến dưới đây sẽ tổ chức những loại màu trộn đang được thực hiện.

```
int currBlend = 4;
```

Chức năng Init của chúng ta sẽ lựa chọn một màu để xóa màn hình. Chúng ta sẽ không sử dụng bất kì chiều sâu nào trong chức năng này.

```
void init()
{
      glClearColor(0.25f, 0.25f, 0.25f, 1.0f);
```

Đê kích hoạt chức năng pha màu chúng ta phải sử dụng cở GL_BLEND của chức năng glEnable.

```
      glEnable(GL_BLEND);
```

Một chức năng quan trọng glBlendFunc được sử dụng để chỉ định màu sắc như thế nào trong việc trộn màu. Chức năng này có 2 giá trị. Xác định màu sắc như thế nào có thể được tính. Cả 2 tham số có thể chấp nhận được các giá trị sau:

- GL_ZERO
- GL_ONE
- GL_SRC_COLOR
- GL_ONE_MINUS_SRC_COLOR
- GL_DST_COLOR
- GL_ONE_MINUS_DST_COLOR
- GL_SRC_ALPHA
- GL_ONE_MINUS_SRC_ALPHA
- GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA

Tham số đầu tiên cũng có thể chấp nhận giá trị của GL_SRC_ALPHA_SATURATE

```
      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
```

Các mảng dưới đây sử dụng để chuyển đổi giữa các ví dụ của việc trộn màu

```
GLenum blendSrc[] = {
      GL_ONE,
      GL_ONE,
      GL_ONE,
      GL_SRC_ALPHA,
      GL_SRC_ALPHA
};

GLenum blendDst[] = {
      GL_ZERO,
      GL_ONE,
      GL_ONE_MINUS_DST_ALPHA,
      GL_ONE,
      GL_ONE_MINUS_SRC_ALPHA
};
```

Chức năng display là nơi 4 hình chữ nhật tạo thành 1 hình vuông

```
 void display()
 {
      glClear(GL_COLOR_BUFFER_BIT);
      glLoadIdentity();

      glVertexPointer(2, GL_FLOAT, 0, rectangle);

      glEnableClientState(GL_VERTEX_ARRAY);

      glPushMatrix();
            glTranslatef(1.5f, 2.0f, 0.0f);
            glColor4f(1.0f, 0.0f, 0.0f, 0.5f);
            glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
      glPopMatrix();

      glPushMatrix();
            glTranslatef(0.7f, 1.5f, 0.0f);
            glRotatef(90.0f, 0.0f ,0.0f, 1.0f);
            glColor4f(0.0f, 1.0f, 0.0f, 0.5f);
            glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
      glPopMatrix();
```

```
      glPushMatrix();
            glTranslatef(1.7f, 1.5f, 0.0f);
            glRotatef(90.0f, 0.0f ,0.0f, 1.0f);
            glColor4f(0.0f, 0.0f, 1.0f, 0.25f);
            glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
      glPopMatrix();

      glPushMatrix();
            glTranslatef(1.5f, 1.0f, 0.0f);
            glColor4f(1.0f, 1.0f, 0.0f, 0.75f);
            glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
      glPopMatrix();

      glFlush();
      glutSwapBuffers();
 }
```

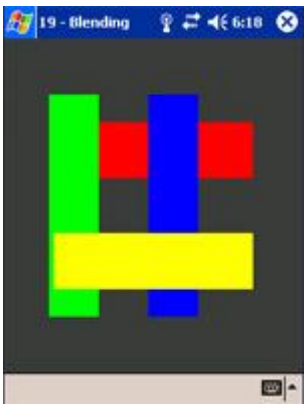
Chúng ta sẽ tạo thêm chức năng menu để xử lí lựa chọn Blending Function và thay đổi việc trộn màu.

```
        case 2 :
            ++currBlend %= 5;
            glBlendFunc(blendSrc[currBlend], blendDst[currBlend]);
            glutPostRedisplay();
            break;
```

Sau khi chạy chương trình bạn có thể thay đổi chế độ trọn màu bằng cách nhấn phím “b”.

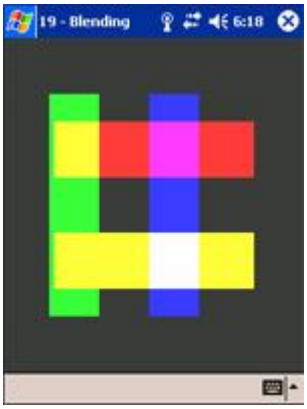
Mặc định của chức năng trộn màu là (GL_ONE, GL_ZERO) điều này làm cho màu sắc không bị pha trộn.

(GL_ONE, GL_ZERO)



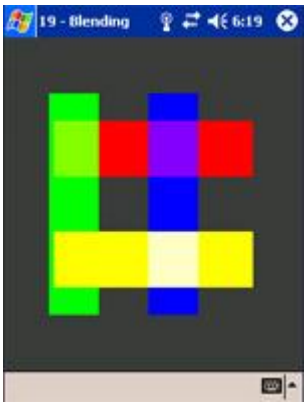
Chức năng trôn màu tiếp theo là (GL_ONE, GL_ONE). Điều này về cơ bản có điểm nguồn và điểm đích của màu sắc và chúng được trộn vào với nhau. Một hỗn hợp của màu xanh lá cây và màu đỏ tạo ra màu vàng như chúng ta nhìn thấy ở góc trên bên trái. Thanh màu vàng có màu xanh lá cây giá trị là 1. Thanh màu xanh lá cây có sự pha trộn của màu vàng nên vẫn giữ nguyên màu vàng. Khi màu vàng được pha trộn với màu xanh lục. Giá trị màu sắc trở thành 1 kết quả hiển thị màu trắng.

(GL_ONE, GL_ONE)



Tiếp theo là chức năng (GL_ONE, GL_ONE_MINUS_DST_ALPHA). Kết hợp những giá trị alpha, như bạn đã nhìn thấy điều này tạo ra 1 chút minh bạch nhưng màu vàng pha trôn với màu xanh lá cây vẫn giữ nguyên màu vàng, màu vàng pha trộn với màu xanh lá cây sẽ ra màu trắng như trước.

(GL_ONE, GL_ONE_MINUS_DST_ALPHA)

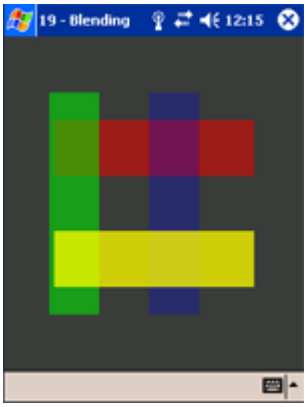


Chức năng trộn kế tiếp là (GL_SRC_ALPHA, GL_ONE), tạo ra một hình tốt hơn, minh bạch hơn. Tất cả các hình chữ nhật giờ đã xuất hiện minh bạch, hình chữ nhật màu vàng giờ bị mờ hơn nó có giá trị alpha là 0.75. Hình chữ nhật màu xanh cũng như vậy nó có giá trị alpha bằng 0.25.

(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)

Mặc dù việc trôn màu đã làm việc tốt, có thể bạn vẫn muốn các hình ở phía dưới hiện rõ hơn điều này đạt được bằng (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) của chức năng trộn màu.

(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)



3.16 Minh bạch đối tượng (Transparency)

Trong phần hướng dẫn trước chúng tôi đã nói về cách làm việc với sự pha màu. Chúng tôi nhận thấy việc làm này rất hữu ích cho việc làm minh bạch đối tượng, trong phần hướng dẫn này sẽ hướng dẫn bạn làm minh bạch một đối tượng 3D.

Hướng dẫn này được sửa đổi từ hướng dẫn 3.13 - Texture Mapping

Nội dung của hàm main.cpp

Bước đầu tiên là bật chức năng trộn màu trong hàm Init

```
glEnable(GL_BLEND);
```

Chức năng pha trôn mà chúng tôi sử dụng là (GL_SRC_ALPHA, GL_ONE)

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

Chúng tôi muốn tất cả các đa giác được vẽ ra bởi vậy chúng tôi vô hiệu hóa chức năng depth testing. Có chiều sâu sẽ làm cho các đa giác ở phía sau không được hiện thị.

```
glDisable(GL_DEPTH_TEST);
```

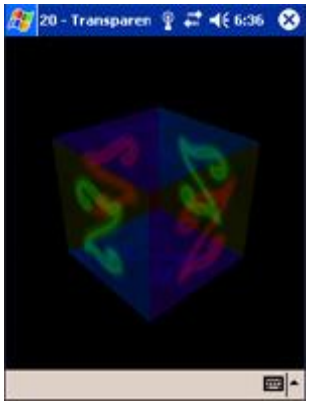
Trong hàm menu.

```
case 3 :
        if (glIsEnabled(GL_BLEND))
        {
                glDisable(GL_BLEND);
                glEnable(GL_DEPTH_TEST);
        }
        else
        {
                glEnable(GL_BLEND);
                glDisable(GL_DEPTH_TEST);
        }
        break;
```

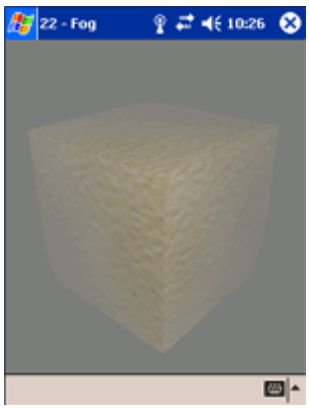
Chúng ta có thể tạo ra một số hiệu ứng tôt. Ta có lại mã kích hoạt hay vô hiệu hóa ánh sáng như trong hướng dẫn 15. Sự khác biệt khi ánh sáng bị vô hiệu hóa sẽ được hiện thị ở dưới đây.

Lighting Enabled

Lighting Disabled



3.17 Hiệu ứng sương mù (Fog)



Hướng dẫn này sẽ bàn về hiệu ứng sương mù, điều này có thể rất hữu ích cho một số các ứng dụng. Hãy tưởng tượng bạn đang tạo ra một trò chơi mà ở đó bạn có thể di chuyển từ quốc gia này đến quốc gia khác, hiển nhiên bạn không thể hiển thị đầy đủ mọi thứ ở 2 nơi điều này sẽ gây ra sự lãng phí, bạn cũng không muốn cảnh vật bất ngờ xuất hiện. Hiệu ứng sương mù được sử dụng để tạo cho bạn cảm giác có một lớp sương mù làm giảm khả năng nhìn của bạn.

Fog cũng làm cho cảnh vật trở lên hiện thực hơn bằng các đặt Fog trong những khu vực có sương mù như là cảnh ở một thung lũng.

Hướng dẫn này được xây dựng theo hướng dẫn 15 - Texture Mapping

Nội dung của hàm main.cpp

Fog có thể có màu sắc cụ thể, bởi vậy chúng tôi tạo ra một mảng để lưu trữ màu sắc này. ở đây chúng tôi sử dụng lớp sương mù màu xám.

```
float fogColor[] = { 0.5f, 0.5f, 0.5f, 1.0f };
```

Có 3 chế độ sương mù:

Cờ	Mô tả
GL_EXP	Đây là dạng đơn giản nhất của Fog, đối tượng không thực sự xuất hiện để di chuyển vào hoặc ra khỏi Fog. Điều này đơn giản chỉ là việc tạo ra một lớp sương mù.
GL_EXP2	Đây là dạng cao hơn của Fog. Đối tượng có thể xuất hiện ngay khi di chuyển vào hoặc ra khỏi Fog. Bạn có thể thấy lớp sương mù không xuất hiện hoàn toàn trong thực tế như là các đường cụ thể khi nhìn vào một điểm nơi mà đối tượng đi chuyển ra khỏi Fog
GL_LINEAR	Đây là cách hiển thi Fog một các thực tế nhất, đối tượng có

	thể vào và ra khỏi Fog một cách đúng cách nhất, đem lại hiệu ứng sương mù một cách tốt nhất.

Mặc dù GL_LINEAR là chế độ sương mù thực tế nhất, điều này không có nghĩa là bạn phải luôn luôn sử dụng nó, rõ ràng các hiệu ứng sương mù tốt hơn sẽ làm chương trình của bạn chạy chậm hơn vậy nên chế độ bạn chọn sẽ phụ thuộc hoàn toàn trên ứng dụng của bạn.

Chúng tôi sẽ tạo một mảng để chứa chế độ sương mù. Cho phép chúng dễ dàng thay đổi. Chúng tôi cũng tạo ra một số nguyên để xác định xem chế độ sương mù nào đang được sử dụng.

```
float fogTypes[] = { GL_EXP, GL_EXP2, GL_LINEAR };
int fogType = 0;
```

Hai chức năng mà bạn sẽ sử dụng nhiều nhất khi làm việc với Fog là chức năng **glFogf** và **glFogfv**. Chức năng fixed như **glFogx** và **glFogxv** là các chức năng có sẵn của OpenGL ES

Chức năng glFogf chấp nhận 2 tham số. Tham số đầu tiên là cờ dùng để xác định các thuộc tính của sương mù được thay đổi. Tham số thứ hai là giá trị float dùng để gán các thuộc tính này.

Bảng dưới đây sẽ miêu tả các thuộc tính của Fog và các giá trị float mà nó sử dụng:

Cờ	Giá trị float	Miêu tả
GL_FOG_MODE	GL_EXP, GL_EXP2 hoặc GL_LINEAR	Xác định chế độ sương mù như đã giải thích ở trên
GL_FOG_DENSITY	> 0.0f (mặc định 1.0f)	Điều này chỉ ra lớp sương mù như thế nào gọi là dày đặc. Giá trị càng cao thì lớp sương mù càng dày đặc
GL_FOG_START	Any float (mặc định	Xác định khoảng cách gần

	0.0f)	nhất của sương mù
GL_FOG_END	Any float (mặc định 1.0f)	Xác định khoảng cách xa nhất của sương mù

Bây giờ chúng ta hiểu làm thế nào để thay đổi thuộc tính của Fog. Chúng ta sẽ thiết lập chế độ sương mù ban đầu tới GL_EXP.

```
bool init()
{
      .
      .
      glFogf(GL_FOG_MODE, GL_EXP);
```

Chức năng glFogfv được sử dụng cho một thuộc tính. Đây là thuộc tính GL_FOG_COLOR nó dùng để miêu tả màu sắc của sương mù. Tham số thứ hai chấp nhận một mảng float, chúng tôi sử dụng mảng màu fogColor cho chức năng này.

```
      glFogfv(GL_FOG_COLOR, fogColor);
```

Tiếp theo chúng ta sẽ thiết lập mật độ sương mù. Thử thay đổi giá trị của nó để thấy được lớp sương thay đổi thế nào.

```
      glFogf(GL_FOG_DENSITY, 0.35f);
```

Một chức năng khác không dành riêng cho Fog là hàm **glHint** chức năng này cho phép bạn xác định những thứ quan trọng như hình/tốc độ của bạn có hiệu lực. Tham số đầu tiên có thể là một số giá trị như GL_FOG_HINT, GL_LINE_SMOOTH_HINT, GL_PERSPECTIVE_CORRECTION_HINT và GL_POINT_SMOOTH_HINT. Chúng tôi chỉ sử dụng cờ GL_FOG_HINT.

Tham số thứ 2 có thể được là GL_DONT_CARE, GL_FASTEST hoặc GL_NICEST nếu chúng ta muốn lớp sương mù của chúng ta là tốt nhất có thể, chúng ta phải sử dụng cờ GL_NICEST. Còn nếu chúng ta qua tâm nhiều hơn đến tốc độ thì ta sử dụng cờ GL_FASTEST. Giá trị mặc định là GL_DONT_CARE. Điều này làm nâng cao hiệu quả của sương mù khi có đủ thời gian để làm việc này.

Chúng tôi sử dụng cờ GL_DONT_CARE cho hàm glHint dưới đây. Điều này không cần thiết vì nó là giá trị mặc đinh, chúng tôi hiển thị nó ở đây để cho bạn biết rằng có thể tăng hiệu quả của sương mù.

```
glHint(GL_FOG_HINT, GL_DONT_CARE);
```

Khoảng cách gần va xa của lớp sương mù

```
glFogf(GL_FOG_START, 1.0f);
glFogf(GL_FOG_END, 5.0f);
```

Như tất cả các phần khác của OpenGL ES chúng ta phải kích hoạt chức năng sương mù thông qua cờ GL_FOG của hàm glEnable.

```
glEnable(GL_FOG);
.
.
}
```

Bây giờ ta có thể kích hoạt và khởi chạy Fog trong hàm **init**, chúng ta có thể hiển thị chúng trên **Display**. Thay vì sử dụng chức năng ugluLookAtf chúng ta có thể dich chuyển ma trận modelview.

```
void display()
{
    .
    .
    glTranslatef(0.0f, 0.0f, -5.0f);
    .
    .
}
```

Phần cuối cùng là thay đổi chức năng menu, khi chế độ Fog được lựa chọn, chúng ta muốn thay đổi chế độ sương mù, điều này được thực hiện bằng các sử dụng mảng fogTypes đã được xác định ở phần đầu của hướng dẫn.

```
case 3 :
    ++fogType %= 3;
    glFogf(GL_FOG_MODE, fogTypes[fogType]);
    break;
```

CHƯƠNG 4: Áp dụng OpenGL ES để tạo ứng dụng đồ họa 3D

4.1 Phát biểu bài toán ứng dụng

Qua tìm hiểu về lý thuyết một số kỹ thuật đồ họa 3D của OpenGL ES, Vận dụng các kiến thức đã thu thập được để sử dụng các kĩ thuật đồ họa 3D trong bộ thư viện mã nguồn mở, làm nền tảng bước đầu cho việc thiết kế một trò chơi 3D trên di động.

Chương trình tạo ra một trò chơi 3D đơn giản nơi bạn có thể điều khiển một chiếc ô tô xung quanh một khoảng đất rộng.

4.2 Một số vấn đề chính và hướng giải quyết

4.2.1 Tạo các file đối tượng đồ họa

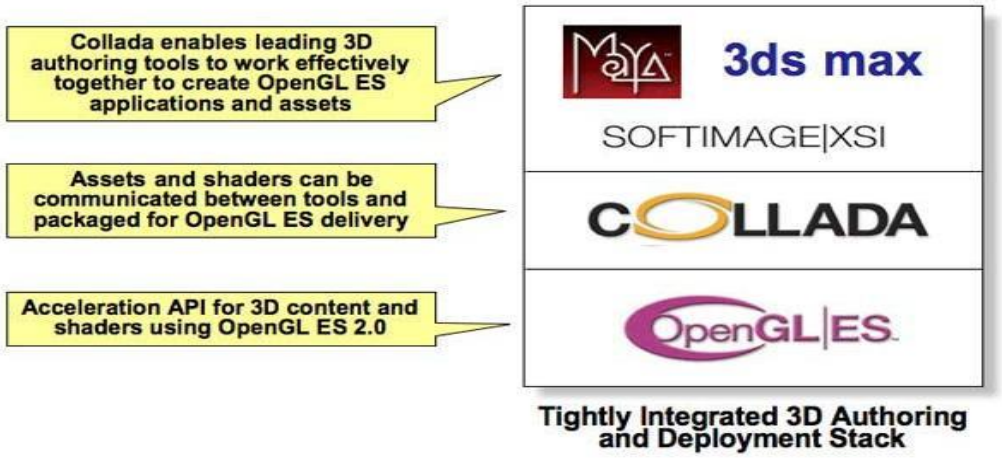
- vấn đề:

Như đã trình bày ở phần OpenGL ES các đối tượng đồ họa được tạo ra từ các mảng chứa các đỉnh của đối tượng. Sẽ là rất khó khăn khi tự tạo ra các mảng này vì một đối tượng có rất nhiều các đỉnh khác nhau.

- Hướng giải quyết:

Ta sẽ sử dụng 1 phần mềm trung gian để tạo ra các đối tượng này như 3DS MAX, MAYA v.v… Và sử dụng bộ công cụ COLLADA để chuyển các đối tượng này thành các file đối tượng .h. Các file đối tượng .h này bao gồm các giá trị như: số đỉnh, số bề mặt và các tọa độ đỉnh cho đối tượng.

https://collada.org



4.2.2 Tạo bản đồ và giới hạn bản đồ

- vấn đề:

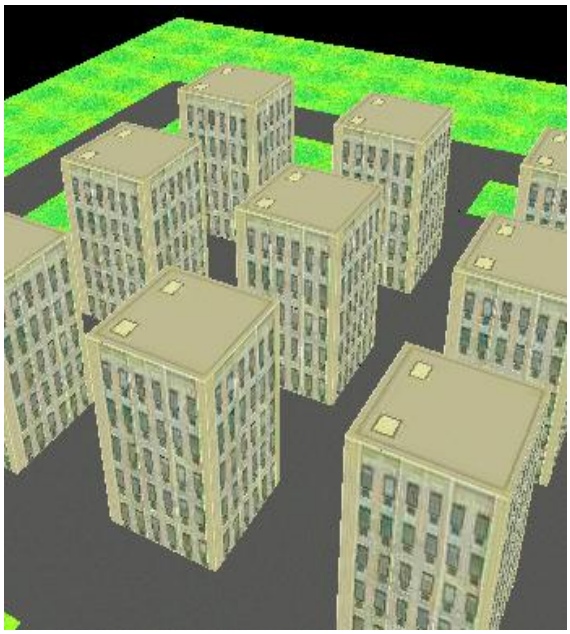
Chắc hẳn việc tạo ra một bản đồ đối với bạn là không mấy khó khăn, nhưng vấn đề ở đây là làm thế nào để giới hạn bản đồ đó lại. Ví dụ như khi bạn điều khiển xe trên đường, tốc độ sẽ nhanh hơn khi bạn đi trên bãi cỏ, hay xe sẽ bị dừng lại khi bạn đâm vào các đối tượng khác như là nhà cửa, hàng rào v.v…

- Hướng giải quyết:

Ta sẽ tạo ra một ma trận tương ứng với bản đồ, và dựa vào ma trận này ta sẽ xác định các đối tượng (như nhà cửa, nền đường, bãi cỏ v.v…) sẽ được đặt ở đâu trên bản đồ và việc giới hạn bản đồ hay xử lí va chạm cũng được thực hiện dựa trên ma trận này.

Đoạn mã minh họa: trong 3Dobject.h

```
const TUint8 KMap[16][32]={
      "xxxxxxxxxxxxxxxx",
      "xxxxxxxxxxxxxxxx",
      "xx00000000000xxx",
      "xx0xxxx0xxxx0xxx",
      "xx0x0000000x0xxx",
      "xx0x0a0a0a0x0xxx",
      "xx0x0000000x0xxx",
      "xx000a0a0a000xxx",
      "xx0x0000000x0xxx",
      "xx0x0a0a0a0x0xxx",
      "xx0x0000000x0xxx",
      "xx0xxxx0xxxx0xxx",
      "xx00000000000xxx",
      "xxxxxxxxxxxxxxxx",
     "xxxxxxxxxxxxxxxx",
      "xxxxxxxxxxxxxxxx",
    };
```



Dựa vào ma trận trên ta sẽ sử dụng các giá tị như sau:

- các giá trị x: đại diện cho bãi cỏ.
- các giá trị 0: đại diện cho đường.
- các giá trị a: đại diện cho nhà.

4.2.3 Xây dựng đối tượng, bắt nút và di chuyển đối tượng.

- Vấn đề:

Ở đây chúng ta sẽ bàn về cách xây dựng các đối tượng, bắt các phím và làm cho chiếc xe của ta di chuyển sao cho tự nhiên nhất. Như việc bánh xe chuyển động (khi di chuyển), quay bánh xe và xe hơi bị nghiêng sang trái hoặc phải khi vào cua, tốc độ của xe sẽ được tăng dần v.v..

- Hương giải quyết:

Khởi tạo các đối tượng 3D từ dữ liệu đã cho

```
// nhà
      i3dHouse = CreateObject( KHouseVertexData, KHouseFaceData,
KNumHouseVertices, KNumHouseFaces, TPoint( 0,0 ), 1 );

// đường
      i3dRoad = CreateObject( KRoadVertexData, KRoadFaceData,
KNumRoadVertices, KNumRoadFaces, TPoint( 0,0 ), 1 );

// bãi cỏ
      i3dGrass = CreateObject( KGrassVertexData, KGrassFaceData,
KNumGrassVertices, KNumGrassFaces, TPoint( 0,0 ), 1 );

// bánh xe
      i3dFrontWheel = CreateWheel( KFrontWheelRadius, 50, 6 );
      i3dRearWheel = CreateWheel( KRearWheelRadius, 150, 6 );

// xe
      i3dCar = CreateObject( KCarVertexData, KCarFaceData,
KNumCarVertices, KNumCarFaces, TPoint( 0,128 ), 10 );
      i3dText = CPolygonObject::NewL( i3DBase, 1, 1 );
```

Thiết lập chất liệu cho mọi đối tượng

```
      i3dHouse->SetTexture( iGlTexture2 );
      i3dRoad->SetTexture( iGlTexture );
      i3dGrass->SetTexture( iGlTexture2 );
      i3dFrontWheel->SetTexture( iGlTexture );
      i3dRearWheel->SetTexture( iGlTexture );
      i3dCar->SetTexture( iGlTexture );
      i3dText->SetTexture( iGlTextureFont );
```

Xây dựng và đặt đối tượng dựa theo ma trận Kmap ở trên

```
for( y=0; y<KMapHeight; y++ )
            {
            TUint8* p = (TUint8*)KMap[ y ];
            for( x=0; x<KMapWidth; x++ )
                  {
                  switch( *p++ )
                        {
                        case '0':
                              {
                              n = iRender->AddObject( i3dRoad );
                              iRender->SetObjectPosition( n,
                                    TVertex( x * KTileSize, 200, y *
KTileSize ) );
                              break;

                        case 'a':
                              {
                              n = iRender->AddObject( i3dHouse );
                              iRender->SetObjectPosition( n,
                                    TVertex( x * KTileSize, 200, y *
KTileSize ) );
                              break;
                              }
                        case 'x':
                              {
                              n = iRender->AddObject( i3dGrass );
                              iRender->SetObjectPosition( n,
                                    TVertex( x * KTileSize, 200, y *
KTileSize ) );
                              break;
                              }

                        }
                  }
}
```

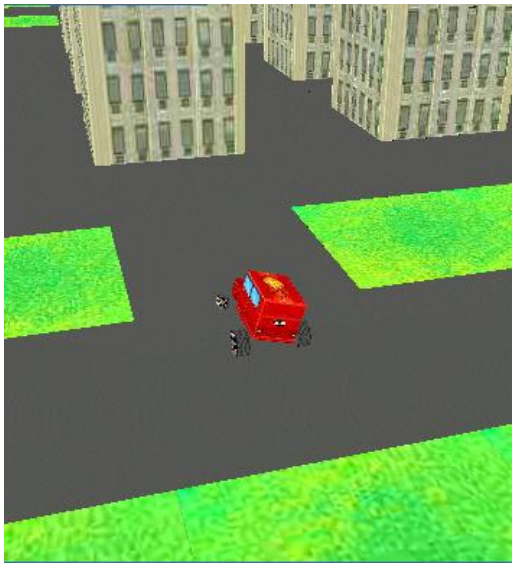
Di chuyển đối tượng thông qua các phím bằng hàm **CGame::Move**

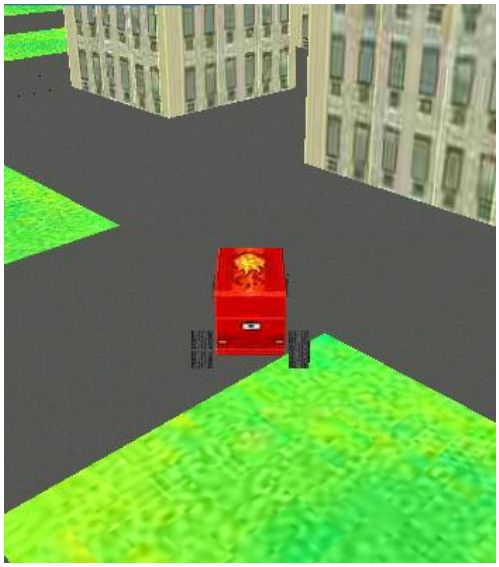
```
void CGame::Move( TUint8* aKey )
	{
	// kiểm rea phím và cập nhập sự thay đổi
	TInt carLastSpeed = iCarSpeed;
	//
	// chuyển động của xe theo hướng chỉ định bởi tốc độ của
xe
	//
	iCarPositionAdd = TVertex( iCarSpeed,0,0 );
	iCarPositionAdd = i3DBase->RotateY( iCarPositionAdd,
iCarAngle & (KSinTableSize-1) );
	iCarPosition += iCarPositionAdd;
	// có rất nhiều các va chạm xuất hiện từ tilemap
	TBool collision = EFalse;
	TInt tileX = ( iCarPosition.iX + KTileSize/2 ) /
KTileSize;
	TInt tileY = ( iCarPosition.iZ + KTileSize/2 ) /
KTileSize;
	// đầu tiên kiểm tra vị trí trên bản đồ:
	if( ( tileX < 1 ) || ( tileX >= KMapWidth-1 ) ||
		( tileY < 1 ) || ( tileY >= KMapHeight-1 ) )
		{
		collision = ETrue;
		}

	// kiểm tra sự va chạm với các tòa nhà
	TInt tile = KMap[ tileY ][ tileX ];
	if( tile == 'a' ) // nếu tile là một tòa nhà
		{
		collision = ETrue;
		}

	if( collision )
		{
		// di chuyển ngược lại
		iCarPosition -= iCarPositionAdd;
		// và dừng
		iCarSpeed = 0;
		}
	//
	// di chuyển vị trí của xe với tốc độ của nó
	// vị trí của xe được sử dụng để làm quay các bánh xe
	//
	iCarPos += iCarSpeed;
	//
	// Diều chỉnh tốc độ của xe thông qua các phím
	// Nhấn phím '1' để tăng tốc
	// Nhấn phím '4' để giảm tốc độ (lùi lại)
	//
```

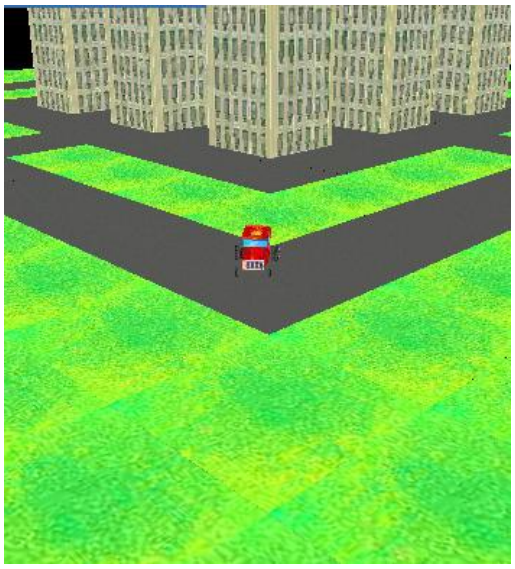
```
      if( aKey[ EStdKeyUpArrow ] || aKey[ '1' ] || aKey[
EStdKeyDevice8 ] )
            {
            iThrottle = 1;
            }
      else if( aKey[ EStdKeyDownArrow ] || aKey[ '4' ] || aKey[
EStdKeyDevice9 ] )
            {
            iThrottle = -1;
            }
      else
            {
            iThrottle = 0;
            }
      //
      // Rẽ xe cùng với bánh lái
      //
      iCarAngleSpeed = ( -iSteerAngle * iCarSpeed ) /
KCarTurnDivider;
      iCarAngle += iCarAngleSpeed;
      // Điều chỉnh bánh lái thông qua các phím
      // Nhấn phím '5' để rẽ sang trái
      // Nhấn phím '6' để rẽ sang phải

      if( aKey[ EStdKeyLeftArrow ] || aKey[ '5' ] || aKey[
EStdKeyDevice6 ] )
            {
            iSteerAngle -= KSteeringSpeed;
            }
      else if( aKey[ EStdKeyRightArrow ] || aKey[ '6' ] || aKey[
EStdKeyDevice7 ]  )
            {
            iSteerAngle += KSteeringSpeed;
            }
      else
            {
            if( iSteerAngle > 0 ) iSteerAngle -= KSteeringSpeed;
            if( iSteerAngle < 0 ) iSteerAngle += KSteeringSpeed;
            }

      //
      // Điều chỉnh tốc độ bởi throttle
      //
      iCarSpeed += iThrottle;
      if( iCarSpeed > 0 )
            {
            iCarSpeed -= iCarSpeed * iCarSpeed / KFriction;
            }
      if( iCarSpeed < 0 )
            {
            iCarSpeed += iCarSpeed * iCarSpeed / KFriction;
            }
```

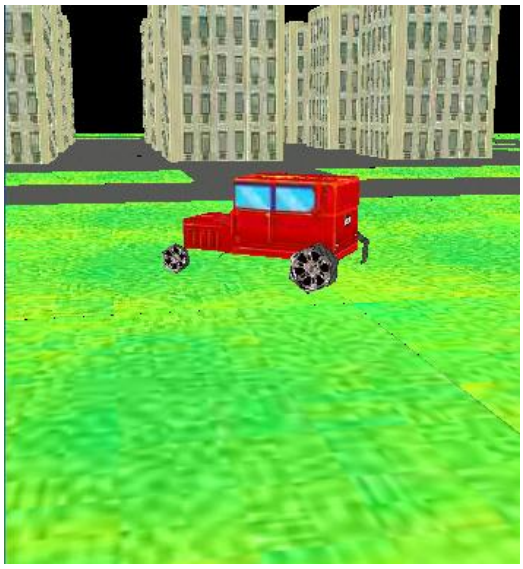
```
      if( iThrottle == 0 )
            {
            if( iCarSpeed > 0 ) iCarSpeed -= 1;
            if( iCarSpeed < 0 ) iCarSpeed += 1;
            }
      //
      // Diều chỉnh bánh lái
      //
      if( iSteerAngle > KSteerLimit ) iSteerAngle = KSteerLimit;
      if( iSteerAngle < -KSteerLimit ) iSteerAngle = -KSteerLimit;

      //
      // Điều chỉnh độ nghiêng của xe khi thay đổi tốc độ
      //
      iCarTilt += ( iCarSpeed - carLastSpeed ) * KCarTiltForce;
      if( iCarTilt > 0 )
            {
            iCarTilt -= ( iCarTilt * iCarTilt /
KCarTiltSpringFactor );
            if( iCarTilt < 0 ) iCarTilt = 0;
            }
      if( iCarTilt < 0 )
            {
            iCarTilt += ( iCarTilt * iCarTilt /
KCarTiltSpringFactor );
            if( iCarTilt > 0 ) iCarTilt = 0;
            }
      }
```

4.3 Một số hình ảnh trong Games





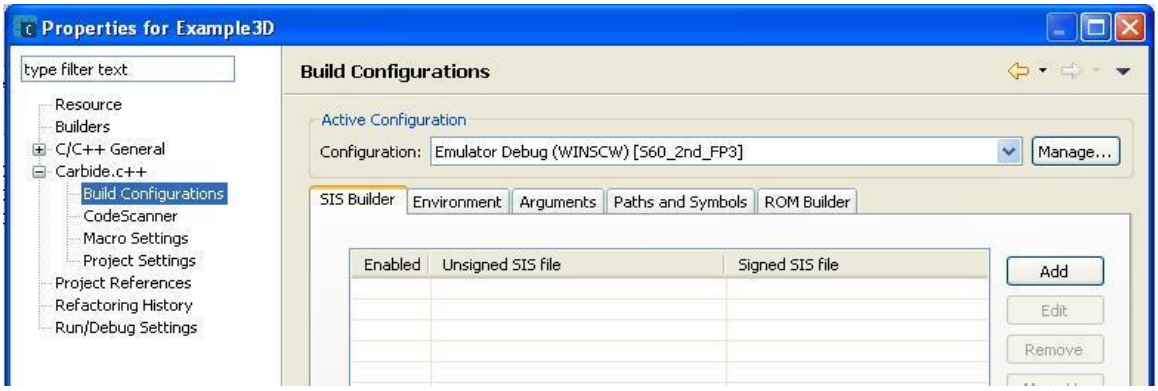




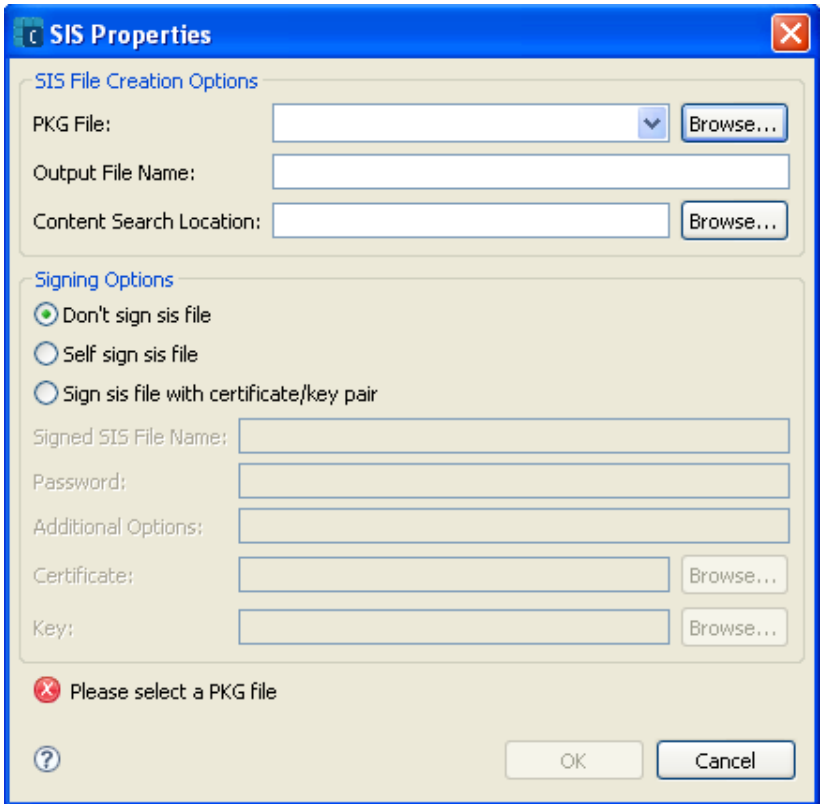
Chương trình chạy trên sdk: **S60 2nd Edition (S60_2nd_ FP3)** được sử dụng cho các dòng máy Nokia 6600, Nokia 6630, Nokia 6680, Nokia N70, Nokia N90, Nokia 9300 Communicator, and Nokia 9500 Communicator v.v…

4.4 Cách tạo file sis để cài đặt lên thiết bị di động

Chọn Project -> Properties -> carbide.c++ -> build configurations



Chọn add -> browse.. -> Chọn file PKG ->ok



Kết luận

Đứng trước sự phát triển của các thiết bị di động và những đòi hỏi của con người trong lĩnh vực giải trí trên các thiết bị di động ngày càng cao. Với đề tài "Ngiên cứu lập trình C++ trên Symbian" khóa luận này đã trình bày được tổng quan về lập trình trên symbian và các kĩ thuật về lập trình đồ họa dựa trên bộ thư viên OpenGL ES. Qua đó xây dựng một mô hình 3D trên Symbian để làm nền tảng cho việc thiết kế các games 3D trên di động.

Trong tương lai em sẽ nghiên cứu thêm về các kĩ thuật đồ họa khác trên thiết bị di động và trên nhiều dòng máy khác nhau. Ngoài ra em sẽ tiếp tục phát triển mô hình trên thành một game 3D hoàn chỉnh, góp phần vào ngành công nghiệp game di động của nước nhà.

Tuy nhiên do hạn chế về điều kiện và thời gian, khoá luận sẽ không thể tránh khỏi những thiếu sót. Kính mong được sự đóng góp ý kiến của thầy cô và các bạn để em có thể hoàn thiện hơn đề tài nghiên cứu của mình trong đợt làm khóa luận tốt nghiệp này.

Trân trọng cảm ơn!

Tài liệu tham khảo

[1]. Symbian OS: 3D Game Engine Example

[2]. OpenGL_ES_Game_Development_-_2004 – Dave Astle, Dave Durnil

[3]. Wiley.Symbian.OS.Internals.Real.time.Kernel.Programming.Dec.2005

[4]. Tìm hiểu và nghiên cứu kĩ thuật phát triển ứng dụng trên môi trường Symbian – Đại học khoa học tự nhiên, hồ chí minh.

[5]. http://wiki.forum.nokia.com/index.php/Category:Symbian_C%2B%2B

[6]. http://wiki.forum.nokia.com/index.php/Category:Symbian_C%2B%2B

[7]. http://www.khronos.org/collada/

[8]. https://collada.org/mediawiki/index.php/Main_Page