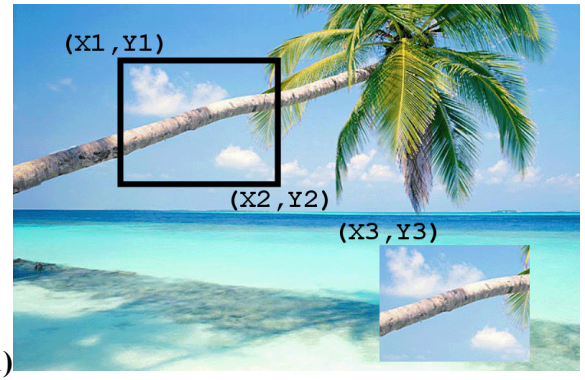
P3 CG 2013.2

	Computação Gráfica 1 Prof. Rodrigo de Toledo	Data: 18/12/2013 P3 2013.2



1) (2 pontos) (adaptado da P1 2012.1)
Faça uma função em C que receba: o ponteiro para um vetor de `unsigned char`, contendo os pixels de uma imagem de tamanho `TAM_X` e `TAM_Y`, no formato RGBRGB, sendo `(0,0)` o seu canto superior esquerdo; e 6 inteiros, representando os pontos `X1, Y1, X2, Y2, X3, Y3`, como na imagem acima. A função deverá copiar os pixels, um a um, da área selecionada para o ponto desejado. Caso não haja espaço para copiar completamente o trecho, copie apenas até a borda e retorne `FALSE`, senão retorne `TRUE`. (Pode considerar que: x1<x2, y1<y2, 0<x1,x2,x3<TAM_X, e 0<y1,y2,y3<TAM_Y).

2) (2 pontos) Quantos bytes ocupam cada uma das seguintes imagens:
a) resolução: 1024 x 512, cores: RGB, 256 cores para cada canal
b) resolução: 256 x 256, 16 cores para cada canal, incluindo o canal alpha para transparência (RGBA)
c) resolução: 2048 x 4096, 1024 cores de uma palheta de cores (desconsidere o tamanho para armazenar a palheta)
d) resolução: 512 x 256, cores: 64 tons de cinza

3) (2 pontos) (P1 2012.2) Qual a ordem correta no pipeline gráfico entre as seguintes etapas?
(a) Clipping; (b) Display/Visibility; (c) Illumination (shading); (d) Modeling Transformations; (e) Projection; (f) Scan Conversion (rasterization); (g) Viewing Transformation.
Continuação: Onde os vertex e fragment shaders se posicionam ao longo desse pipeline?

4) (2 pontos) (parte da P2 2012.2) Escreva uma função float areaTotal (Vertex *v) que retorne a área total dos triângulos ao redor de um vértice v, que não pertence a borda, em uma malha de estrutura half-edge. Considere que há um tipo vec2D com operações básicas, como a subtração de dois pontos e produtos escalar/vetorial.
class Vertex {Point2D p; H_Edge hEdge;} class H_Edge { Vertex vOrig; H_Edge eTwin; Face f; H_Edge eNext;} class Face { H_Edge HEdge;}

5) (2 pontos)
Dado o código da função que retorna a interseção entre o raio e a difrença de duas esferas (CSG, A – B). Inclua o cálculo da normal normalizada no ponto de interseção nesse código.

```
bool SpheresDiferenceIntersection (vec3 ray_origin, vec3 ray_direction, float radiusA, vec3 centerA,
 float radiusB, vec3 centerB, float *t1, vec3 *normal){
        float t1A, t2A, t1B, t2B;
        if ( ! intersection(ray_origin,  ray_direction,  radiusA,  centerA,  &t1A,  &t2A) )
                return false;
        if ( ! intersection(ray_origin,  ray_direction,  radiusB,  centerB,  &t1B,  &t2B) )
        {        *t1 = t1A;        return true;        }
        if (t2B < t1A  ||  t1A < t1B)
        {        *t1 = t1A;        return true;        }
        if (t2B > t2A)
                return false;
        *t1 = t2B;
        return true;
}
```

Gabarito:
1)

a) 1.5 MB

b) 2^8 x 2^8 x 2^2 x 2^2 bits = 2^{20} bits = 2^{17} Bytes = 128 KB

c) 2^{11} x 2^{12} x 10 bits = 10 x 2^{23} bits = 10 x 2^{20} Bytes = 10 MB

d) 2^9 x 2^8 x 6 bits = 6 x 2^{17} bits = 6 x 2^{14} Bytes = 96 KB

3) (1) D C G E A F B, ou: (2) D C G A E F B
Vertex shader: substitui DCGE para sequência (1) ou DCG para (2)
Fragment shader: entre F e B ((respota correta também se for F))

5)
Trocar

```
{*t1 = t1A;         return true;          }
```

por

```
{*t1 = t1A;         *normal = (ray_origin + (ray_direction * *t1) -centerA) / radiusA;          return
true;          }
```

Após

```
*t1 = t2B;
```

incluir

```
*normal = (centerB - (ray_origin + (ray_direction * *t1))) / radiusB;
```